Malu Castellanos
Umeshwar Dayal
Wolfgang Lehner (Eds.)

LNBIP 126

Enabling Real-Time Business Intelligence

5th International Workshop, BIRTE 2011
Held at the 37th International Conference
on Very Large Databases, VLDB 2011
Seattle, WA, USA, September 2011, Revised Selected Papers

Springer

Lecture Notes
in Business Information Processing 126

Series Editors

Wil van der Aalst
Eindhoven Technical University, The Netherlands
John Mylopoulos
University of Trento, Italy
Michael Rosemann
Queensland University of Technology, Brisbane, Qld, Australia
Michael J. Shaw
University of Illinois, Urbana-Champaign, IL, USA
Clemens Szyperski
Microsoft Research, Redmond, WA, USA

Malu Castellanos
Umeshwar Dayal
Wolfgang Lehner (Eds.)

Enabling Real-Time Business Intelligence

5th International Workshop, BIRTE 2011
Held at the 37th International Conference
on Very Large Databases, VLDB 2011
Seattle, WA, USA, September 2, 2011
Revised Selected Papers

Springer

Volume Editors

Malu Castellanos
Hewlett-Packard
Palo Alto, CA, USA
E-mail: malu.castellanos@hp.com

Umeshwar Dayal
Hewlett-Packard
Palo Alto, CA, USA
E-mail: umeshwar.dayal@hp.com

Wolfgang Lehner
Dresden University of Technology
Dresden, Germany
E-mail: wolfgang.lehner@tu-dresden.de

ISSN 1865-1348 e-ISSN 1865-1356
ISBN 978-3-642-33499-3 e-ISBN 978-3-642-33500-6
DOI 10.1007/978-3-642-33500-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012947035

ACM Computing Classification (1998): H.3, J.1, H.2

© Springer-Verlag Berlin Heidelberg 2012
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Business intelligence has evolved into a multi-billion dollar market over the last decade. Since the early years of data warehousing, business needs have constantly posed new requirements on state-of-the-art business intelligence systems. In today's competitive and highly dynamic environment, providing insight does not merely require analysis of the existing data. Deriving actionable intelligence demands the efficient processing of a vast amount of information, in order to arrive at a timely representation of the state of an enterprise as well as of emerging trends. Prediction models must be used in order to assist with the derivation of actions from the current state of the enterprise and the market, taking into account the uncertainty of the prediction. Moreover, the increasing use of twitter, blogs, and other media means that business intelligence cannot restrict itself to only dealing with structured information. More and more information sources of varying kind have to be integrated, starting with the vast amount of textual information in corporate intranets and the Web. However, because of media convergence, future business intelligence will also have to consider audio and video streams as further information sources. The end goal is to support better and timelier decision making, enabled by the availability of up-to-date, high-quality information.

Although there has been progress in this direction and many companies are introducing products toward meeting this goal, there is still a long way to go. In particular, the whole life cycle of business intelligence requires new techniques and methodologies capable of dealing with the new requirements imposed by the new generation of business intelligence applications. From the capturing of real-time business data to the transformation and delivery of actionable information, all the stages of the business intelligence cycle call for new algorithms and paradigms as the basis of new functionalities including business intelligence over text data, ensuring information quality, dealing with the uncertainty of prediction models, nested complex events, and optimizing complex ETL workflows, just to name a few.

The series of BIRTE workshops aims to provide a forum to discuss and advance the foundational science and engineering required to enable real-time business intelligence and the novel applications and solutions that build on these foundational techniques. Following the success of our previous workshops colocated with the VLDB conferences in Seoul, Auckland, Lyon, and Singapore, our fifth workshop was held in Seattle on September 2, 2011.

After the official opening of the workshop, Guy Lohman (IBM Almaden Research Center, USA) delivered an excellent and engaging keynote entitled "Blink: Not Your Father's Database," where he outlined the key concepts of the IBM Accelerator for analytical database queries. The first session on "Innovative System Architectures" included two research papers. The contribution by Qiming Chen

et al. (HP Labs, USA) outlined the core concepts of MemcacheSQL, a SQL Cache engine for massive scale-out query processing. The second paper presented by Florian Huebner with co-authors from Hasso Plattner Institute and SAP proposed a cost-aware strategy for merging differential stores in column-oriented in-memory DBMS. The session was closed with an invited talk delivered by Jose Blakeley from Microsoft Corporation on the architecture of the Microsoft SQL Server Parallel Data Warehouse. In this presentation Jose described how the system exploits parallelism to provide a foundation for real-time enabled enterprises. The second session on Novel Query Support included two research papers. "Relax and Let the Database do the Partitioning Online" by Alekh Jindal and Jens Dittrich from Saarland University, Germany and "Adaptive Processing of Multi-Criteria Decision Support Queries" by Venkatesh Raghavan et al. (Worcester Polytechnic Institute, USA). The final session on "Innovative Applications" again was a combination of an invited talk and the presentation of a research paper. The invited talk was delivered by Shilpa Lawande and Lakshmikant Shrinivas from Vertica, a Hewlett Packard Company on "Scalable Social-Graphing Analytics with the Vertica Analytic Platform." They gave an impressive demo on how suitable modern database platforms such as Vertica are for non-standard applications like graph processing on extremely large databases. The final presentation within the application part of the workshop was delivered by Amit Rustagi (eBay), and outlined the requirements and architecture of "A Near Real-Time Personalization for eCommerce Platform."

We wish to express special thanks to the Program Committee members for helping us prepare an interesting program. To our keynote speakers, presenters, and attendees, we express our appreciation for sharing their work and the lively discussions that made this workshop a great forum for exchanging new ideas. We thank the VLDB 2011 organizers for their help and organizational support. Finally, we would like to thank Maik Thiele, our Publication Chair.

September 2012

Malu Castellanos
Umeshwar Dayal
Wolfang Lehner

Organization

Organizing Committee

General Chair

Umeshwar Dayal	Hewlett-Packard, USA

Program Committee Chairs

Malu Castellanos	Hewlett-Packard, USA
Wolfgang Lehner	TU Dresden, Germany

Program Committee

Christof Bornhövd	SAP Labs, USA
Ben Chin Ooi	National University of Singapore, Singapore
Chetan Gupta	HP Labs, USA
Howard Ho	IBM, USA
Alfons Kemper	TUM, Germany
Christian Konig	Microsoft, USA
Alexander Loeser	University of Technology Berlin, Germany
Jose Norberto Mazon	University of Alicante, Spain
Renee Miller	University of Toronto, Canada
Torben B. Pedersenv	Aalborg University, Denmark
Elke Rundensteiner	WPI, USA
Donovan Schneider	SalesForce, USA
Eric Simon	SAO-BO, France
Nesime Tatbul	ETH Zurich, Switzerland
Florian Waas	Greenplum, USA

Publication Chair

Maik Thiele	Technische Universität Dresden, Germny

Table of Contents

Blink: Not Your Father's Database! 1
Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy Lohman, Konstantinos Morfonios, Keshava Murthy, Lin Qiao, Vijayshankar Raman, Sandor Szabo, Richard Sidle, and Knut Stolze

MemcacheSQL – A Scale-Out SQL Cache Engine 23
Qiming Chen, Meichun Hsu, and Ren Wu

A Cost-Aware Strategy for Merging Differential Stores in Column-Oriented In-Memory DBMS 38
Florian Hübner, Joos-Hendrik Böse, Jens Krüger, Cafer Tosun, Alexander Zeier, and Hasso Plattner

Microsoft SQL Server Parallel Data Warehouse: Architecture Overview 53
José A. Blakeley, Paul A. Dyke, César Galindo-Legaria, Nicole James, Christian Kleinerman, Matt Peebles, Richard Tkachuk, and Vaughn Washington

Relax and Let the Database Do the Partitioning Online 65
Alekh Jindal and Jens Dittrich

Adaptive Processing of Multi-Criteria Decision Support Queries 81
Shweta Srivastava, Venkatesh Raghavan, and Elke A. Rundensteiner

Scalable Social Graph Analytics Using the Vertica Analytic Platform ... 98
Shilpa Lawande, Lakshmikant Shrinivas, Rajat Venkatesh, and Stephen Walkauskas

A Near Real-Time Personalization for eCommerce Platform 109
Amit Rustagi

Author Index 119

Blink: Not Your Father's Database!

Ronald Barber[1], Peter Bendel[2], Marco Czech[3,*], Oliver Draese[2], Frederick Ho[4], Namik Hrle[2], Stratos Idreos[5], Min-Soo Kim[6], Oliver Koeth[2], Jae-Gil Lee[7], Tianchao Tim Li[2], Guy Lohman[1], Konstantinos Morfonios[8], Rene Mueller[1], Keshava Murthy[4], Ippokratis Pandis[1], Lin Qiao[9], Vijayshankar Raman[1], Sandor Szabo[2], Richard Sidle[1], and Knut Stolze[2]

[1] IBM Almaden Research Center, K55/B1, 650 Harry Rd., San Jose, CA 95210
{Rjbarber,lohmang,muellerr,ipandis,ravijay,rsidle}@us.ibm.com
[2] IBM Germany Research and Development Lab, Böblingen, Germany
{peter_bendel,draese,hrle,okoeth,tianchao,
szabo,stolze}@de.ibm.com
[3] SIX Group, Zurich, Switzerland
marco.czech@gmx.de
[4] IBM Information Management Development, 4400 North First St.,
Ste. 100, San Jose, CA 95134
{hof,rkeshav}@us.ibm.com
[5] CWI, Amsterdam, Netherlands
S.Idreos@cwi.nl
[6] DGIST, Daegu, KOREA 711-873
minsoo.k@gmail.com
[7] Korea Advanced Institute of Science and Technology (KAIST),
Dept. of Knowledge Service Engineering, Daejeon, KOREA 305-701
jaegil@kaist.ac.kr
[8] Oracle, 400 Oracle Parkway, Redwood Shores, CA 94065
konstantinos.morfonios@oracle.com
[9] LinkedIn, 2029 Stierlin Court, Mountain View, CA 94043
lqiao@yahoo.com

Abstract. The Blink project's ambitious goals are to answer all Business Intelligence (BI) queries in mere seconds, regardless of the database size, with an extremely low total cost of ownership. It takes a very innovative and counter-intuitive approach to processing BI queries, one that exploits several disruptive hardware and software technology trends. Specifically, it is a new, workload-optimized DBMS aimed primarily at BI query processing, and exploits scale-out of commodity multi-core processors and cheap DRAM to retain a (copy of a) data mart completely in main memory. Additionally, it exploits proprietary compression technology and cache-conscious algorithms that reduce memory bandwidth consumption and allow most SQL query processing to be performed on the compressed data. Ignoring the general wisdom of the last three decades that the only way to scalably search large databases is with indexes, Blink always performs simple, "brute force" scans of the entire data mart in parallel on all nodes, without using any indexes or

[*] Work done while the author was at IBM.

M. Castellanos, U. Dayal, and W. Lehner (Eds.): BIRTE 2011, LNBIP 126, pp. 1–22, 2012.
© Springer-Verlag Berlin Heidelberg 2012

materialized views, and without any query optimizer to choose among them. The Blink technology has thus far been incorporated into two products: (1) an accelerator appliance product for DB2 for z/OS (on the "mainframe"), called the IBM Smart Analytics Optimizer for DB2 for z/OS, V1.1, which was generally available in November 2010; and (2) the Informix Warehouse Accelerator (IWA), a software-only version that was generally available in March 2011. We are now working on the next generation of Blink, called BLink Ultra, or BLU, which will significantly expand the "sweet spot" of Blink technology to much larger, disk-based warehouses and allow BLU to "own" the data, rather than copies of it.

Keywords: Business Intelligence, database management system, query processing, main-memory, multi-core, data mart, OLAP, accelerator, appliance, compression, memory bandwidth, cache-friendly algorithms, data dictionary, encoded data.

1 Introduction

1.1 Motivation

The problem with today's processing of Business Intelligence (BI) queries is that performance is too unpredictable. When an analyst submits a BI query, she doesn't know whether it will run for a few seconds or a few days. If the right performance layer of indexes and materializations (e.g., Materialized Query Tables (MQTs)) has properly anticipated that query, it may run in seconds, whereas without the right indexes or MQTs, the query may take hours or even days, depending upon the data volumes. But BI querying is inherently *ad hoc* – an analyst will typically formulate her next query based upon the results of the previous query. So it's almost impossible to anticipate the workload by looking at past queries, and hence to anticipate the performance layer that will handle all possible queries the analyst might submit in the future.

The goal of Blink is to rectify this high variance in the response time of BI queries and achieve predictably and almost uniformly fast *ad hoc* querying, so that any query will run in approximately the same small time – in seconds or tens of seconds – a timeframe that permits the analyst to truly interact with the data.

1.2 Scope

The target market for Blink is the Business Intelligence (BI) query market, often called OLAP (On-Line Analytics Processing). BI queries are often quite complex, *ad hoc* queries that typically are looking for trends or exceptions in the data, in order to make actionable business decisions. They usually access a large subset of the database, unlike On-Line Transaction Processing (OLTP) workloads, which access only a handful of rows. In order for a human to be able to see trends or exceptions, the large volume of data must be summarized, by grouping the data along some dimension (or dimensions) and aggregating one or more metrics with functions like COUNT or SUM or AVERAGE. This sort of query is the "sweet spot" of Blink.

Databases in this market are typically characterized by a "star" or "snowflake" schema, referring to the shape of the schema graph, as shown in Figure 1. At the center of the star or snowflake is typically the largest table in the schema, called the

"fact table", containing millions to hundreds of billions of rows. Often the fact table contains information about sales transactions, where each row has information about a particular transaction, such as "metrics" like the sales price, and various "dimensions" that characterize that transaction, such as the date and time of the sale, the product sold, and the geographical location of the store that sold it. To save space and avoid redundancy that would be expensive to update, the fact-table rows store only a foreign key to the full dimensional data, which is stored in dimension tables. Those dimensions can, in turn, have dimensions (often forming a hierarchy), forming a schema that more resembles a snowflake than a star.

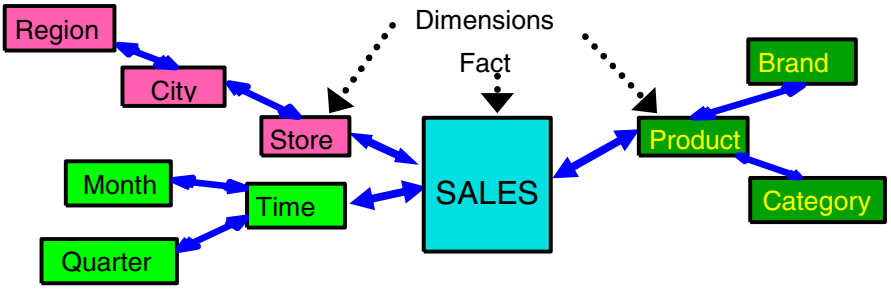


Fig. 1. Example (simplified) "snowflake schema" for a typical Business Intelligence query, composed of a large fact table and multiple dimensions

```
SELECT P.Manufacturer, S.Type, SUM(Revenue)
FROM Fact_Sales F
        INNER JOIN Dim_Product P ON F.FKP = P.PK
        INNER JOIN Dim_Store S ON F.FKS = S.PK
        LEFT OUTER JOIN Dim_Time T ON F.FKT = T.PK
WHERE P.Type = 'JEANS'
    AND S.Size > 50000
    AND T.Year = 2007
GROUP BY P.Manufacturer, S.Type
```

Fig. 2. Example (simplified) SQL for a typical Business Intelligence query for the above snowflake schema, using the INNER JOIN and LEFT OUTER JOIN syntax

If you speak SQL, Figure 2 provides a very simple example BI query that illustrates the previous ideas in a more concrete way on the schema of Figure 1. The fact table Fact_Sales is being joined with three dimension tables, Dim_Product, Dim_Store, and Dim_Time. We use the INNER JOIN syntax here to emphasize that it's a join, but you can also just list the tables in the FROM-list, and provide the join predicates in the WHERE-clause. Note that Blink supports left outer joins as well, but only those in which the fact table is the preserved side and the dimension is the null-producing side. This permits aggregations that include fact-table rows having

unknown values for some dimensions (in this case, for time). Each dimension is limited by a selection predicate in this example, limiting our query to jeans products sold in stores of size greater than 50,000 square feet in the year 2007. This query is grouping the resulting rows by the product manufacturer and store type, and summing the revenues for each manufacturer-type pair.

2 Blink Overview and Architecture

Blink is an accelerator technology that caches a compressed copy of a portion of the data warehouse in main memory and exploits the hardware of modern commodity multi-core processors and proprietary compression techniques to speed up query processing, typically by orders of magnitude compared to conventional DBMSs, without having to "tune" the queries by selecting the best indexes, materialized views, or other performance enhancers. This technology has already been exploited in two IBM products that are generally available (GA): the IBM Smart Analytics Optimizer for DB2 for z/OS V1.1 (GA'd Nov. 2010), a network-attached accelerator to DB2 for z/OS that runs on the zEnterprise Blade eXtension (zBX), which is network-attached to a zEnterprise (mainframe) system [IBM 10]; and also a software-only accelerator to Informix called the Informix Warehouse Accelerator (GA'd March 2011).

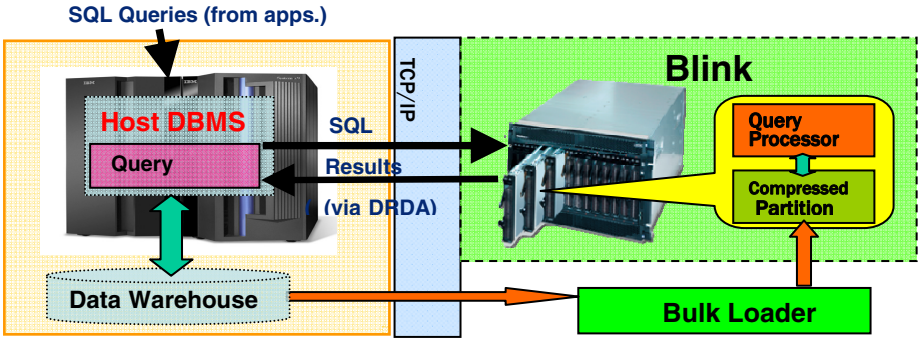


Fig. 3. Architecture of Blink as an accelerator appliance to a host database such as DB2 for z/OS or Informix

Blink is (currently) configured as an accelerator appliance that is network-attached to the host DBMS (DB2 for z/OS or Informix), as shown in Figure 3. We start with a standard data warehouse managed by the host DBMS, and add an instance of Blink that runs on either the zEnterprise Blade eXtension (zBX), which is connected to the z platform via TCP/IP (in the DB2 case), or, with Informix, on either a blade server or the same physical hardware as the database server. However, the user doesn't really see this accelerator – there are no externalized interfaces to the accelerator. Once the user has defined the data of interest to be accelerated, a bulk loader extracts a copy of that data automatically from the data warehouse, pipes the data to the accelerator, analyzes it, and compresses it for storage on the blades of the accelerator. The assignment of data to individual blades is arbitrary and not controllable by the user. Once the data has been so loaded, SQL queries coming into the host DBMS that

reference that data may be routed automatically by the host optimizer to Blink, where it will be executed on the accelerator's compressed data, rather than the host DBMS. The SQL query is first parsed and semantically checked for errors by the host DBMS before being sent to the accelerator via the DRDA protocol in a pre-digested subset of SQL, and results are returned to the host DBMS, and thence to the user, via DRDA, or any host DBMS protocol. Note that the user need not make *any* changes to their SQL queries to get the router to route the SQL query to the accelerator – it simply has to reference a subset of the data that has been loaded.

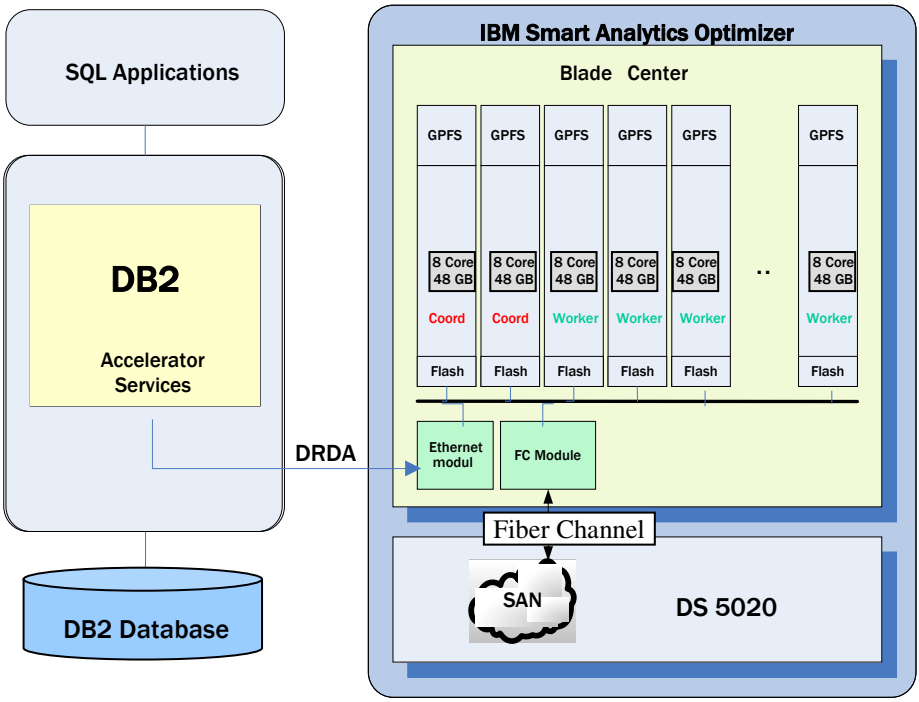


Fig. 4. Hardware architecture of the IBM Smart Analytics Optimizer for DB2 for z/OS, V1.1, showing the contents of the zEnterprise Blade Extension (zBX) and its IBM DS 5020 backup storage

2.1 IBM Smart Analytics Optimizer for DB2 for z/OS

Looking a little closer at the cluster architecture of Blink in the IBM Smart Analytics Optimizer for DB2 for z/OS [SBS+ 11], and the zBX Blade Center H on which it runs, we see in Figure 4 an example of the "Small" configuration, with one fully-populated Blade Center containing 14 blades. (Blink comes in 5 "T-shirt" sizes, "Extra-Small" (1/2 Blade Center), "Small" (1 Blade Center), "Medium" (2 Blade Centers), "Large" (3 Blade Centers), and "Extra-Large" (4 Blade Centers).) All blades contain two sockets, with a quad-core Nehalem chip in each socket, for a total of 8 cores per blade, and 48 GB of real DRAM and some flash memory used for storing intermediate results. Blades are (soft-) designated as either coordinators or workers.

Coordinator blades receive queries from DB2 and broadcast them to the workers, then receive partial answers from the workers, merge the results, and return them to DB2. Only worker blades store data that has been replicated from DB2, and dedicate up to 32 GB of DRAM for storing data; the rest is working memory used for storing the system code and intermediate results. A DS 5020, connected to the Blade Center via the General Purpose File System (GPFS) [SH 02], is needed only to back up the accelerator system code and the compressed data of each worker node – no data is accessed on the DS 5020 during normal query operation (remember, the accelerator is a main-memory database). There are always at least 2 active coordinator blades to avoid a single point of failure, plus one held in reserve that can take over for any worker blade that might fail by simply loading its image from the DS 5020. While waiting for such an unlikely failure, however, the reserve blade can also act as a coordinator. Since IBM Blade Centers when fully populated hold 14 blades, there can be 11 worker blades per Blade Center. This means that one IBM Blade Center can hold at most 11 * 32 GB = 352 GB of data. Since Blink can achieve compression ratios of 2x to 8x, or even more, depending upon the data, we can conservatively estimate a simple rule of thumb that each Blade Center can hold at least 1 TB of raw (pre-load) data, and probably a good deal more. This means that the maximum configuration of the "Extra-Large" configuration can hold at least 4 TB of raw data in memory.

2.2 Informix Warehouse Accelerator

Blink technology is also available via a software offering, called the Informix Ultimate Warehouse Edition (IUWE). The Blink engine, named Informix Warehouse Accelerator (IWA), is packaged with the Informix database server as a main-memory accelerator to it, as shown in Figure 5. IWA is available on the following platforms: the Linux operating system on Intel processor–based servers; the IBM AIX® operating system on IBM POWER7® processor–based servers; the HP-UX operating system on Intel Itanium processor–based servers; and the Oracle Solaris operating system on Oracle SPARC servers. When running Informix database server and Informix Warehouse Accelerator on Linux, the database server and the accelerator can be installed on the same or different computers.

Informix database server and IWA communicate via TCP/IP. When both Informix and IWA are running on the same machine, the coordinator and worker nodes simply become processes on the same machine that communicate via loopback. Informix and IWA can be on distinct SMP hardware, with IWA running both the coordinator and worker processes on the same hardware. IWA can also be deployed on a blade server for increased capacity and performance, such as Intel Xeon-based IBM servers supporting up to 80 cores and 6 TB of DRAM. Nodes on IBM blade servers support up to 4 sockets and up to 640 GB of DRAM. Since both the number of cores and memory capacity of the hardware is increasing rapidly, the IUWE software was packaged flexibly enough to run directly on hardware or in a virtualized/cloud environment. Each database server can have zero, one, or more IWAs attached to it.

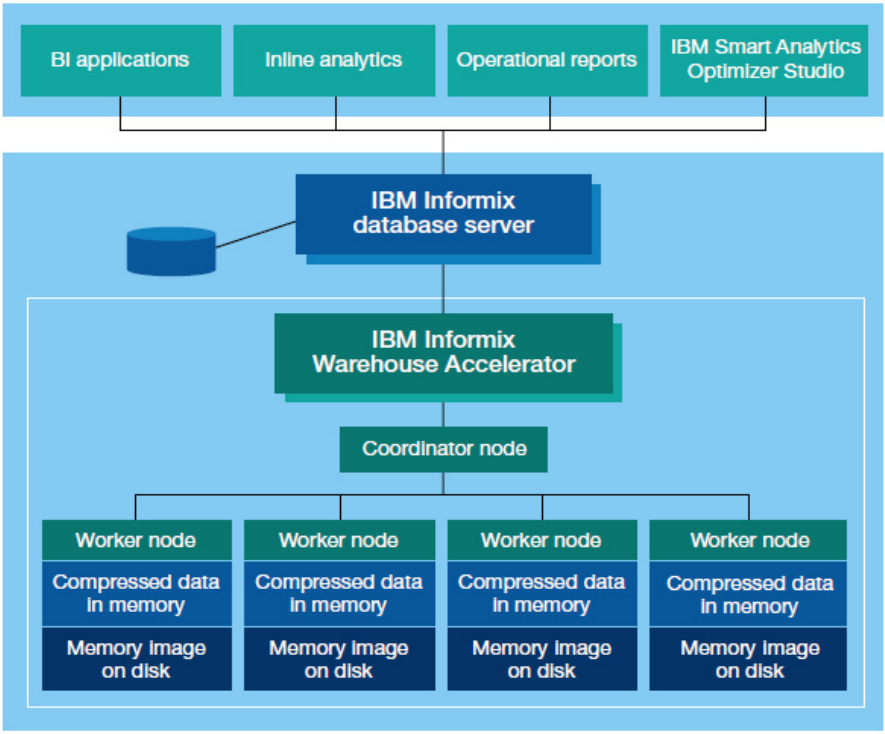


Fig. 5. Architecture of IBM Informix Warehouse Accelerator

3 Performance

Blink is all about query performance. So how good is the performance? Figure 6 shows the absolute run times in seconds (on the y-axis at the left) of queries from our alpha customer, run both with the IBM Smart Analytics Optimizer (abbreviated hereafter as ISAO) and without (i.e., on a traditional disk-based DBMS). The times without ISAO are shown in blue, and with ISAO are shown in red. Since some run times (especially for ISAO) are hard to see, we have labeled the top of each bar with the exact time, in seconds. In addition, we show the speed-up ratio – defined as the ratio of the non-ISAO time to the ISAO time – in green bars and measured on the y-axis to the right. Performance experts for the traditional DBMS chose the best indexes for, and tuned, each query run without ISAO, in consultation with the customer. Remember that running without ISAO is on a traditional, disk-based DBMS, whereas ISAO has all the data in memory.

On average, ISAO sped up this set of user queries by an average of 85 times! However, the key thing to notice from this chart is that the speed-up ratios are proportional to the run time without ISAO, i.e., was greatest for the queries that ran the longest without ISAO. This means that ISAO accelerated the problem queries the most. This is a DBA's dream!

How could this be? Figure 7, which zooms in on just the ISAO times (in seconds), reveals the secret – the denominator of the speed-up ratio is almost constant. All queries completed in ISAO in 11 to 26 seconds, depending upon the complexity of the query. This seems remarkable, as one of the queries (Q5200) ran without ISAO in about an hour and a half, until you realize that ISAO uses essentially the same simple scan plan for every execution of any query. So in this way, execution times vary only depending upon the number of tables, the number of predicates, the selectivity of those predicates, the number of GROUP BY columns, and any ORDER BY sorting.

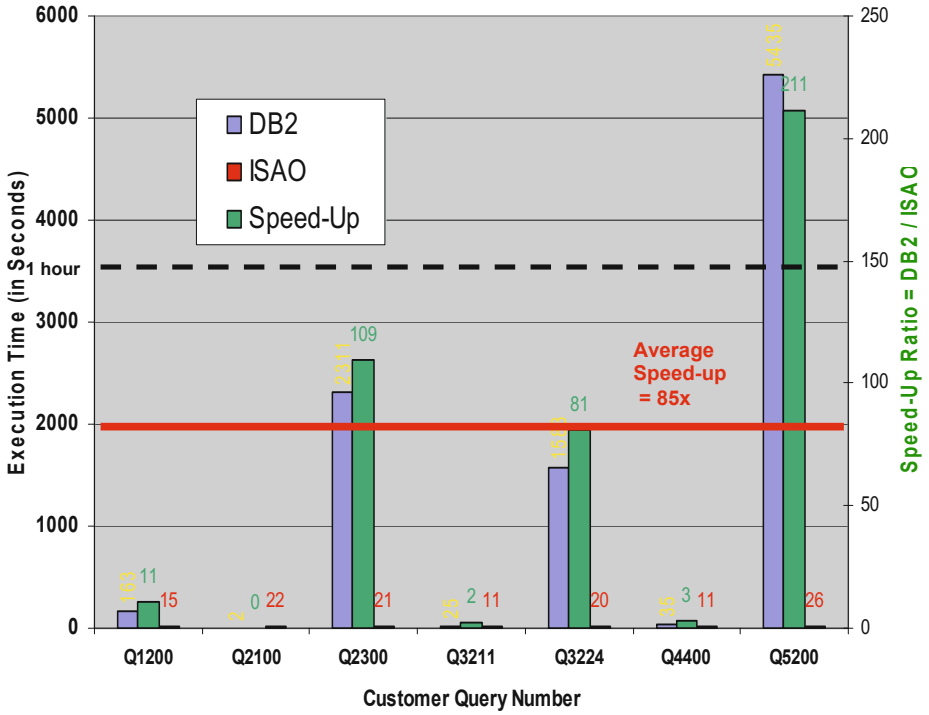


Fig. 6. Relative performance of the IBM Smart Analytics Optimizer product based upon Blink versus a traditional disk-based DBMS, and the resulting speed-up ratio, for an example customer workload

Figure 7 also reveals that performance improvements made to ISAO by October 2009 improved the execution times to run all queries in just 3 to 8 seconds, an improvement of over 3x from our earlier results! All, that is, except query Q2100, which is missing an ISAO execution time in these later results. Why? The December 2008 results shown in Figure 6 indicate that query Q2100 actually ran faster on the traditional, disk-based DBMS (only 2 seconds, vs. 22 on ISAO). So it would be better to not route it to ISAO, because this particular query is more of a "point query" that touches little data and probably uses the perfect index for that query. The DB2 for z/OS Optimizer estimates both execution times as part of its optimization, so it can – and should – avoid routing such queries to ISAO that do not fit our "sweet spot".

We fixed this so that the DB2 Optimizer now makes this determination, and so when we reran the alpha customer's workload in October 2009, DB2 correctly never routed this query to ISAO.

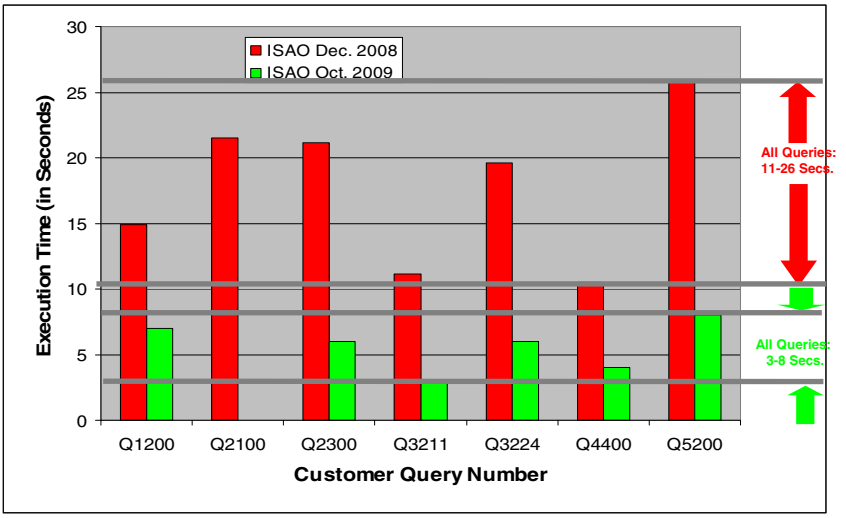


Fig. 7. IBM Smart Analytics Optimizer (ISAO) performance improvement for the customer workload of Figure 6

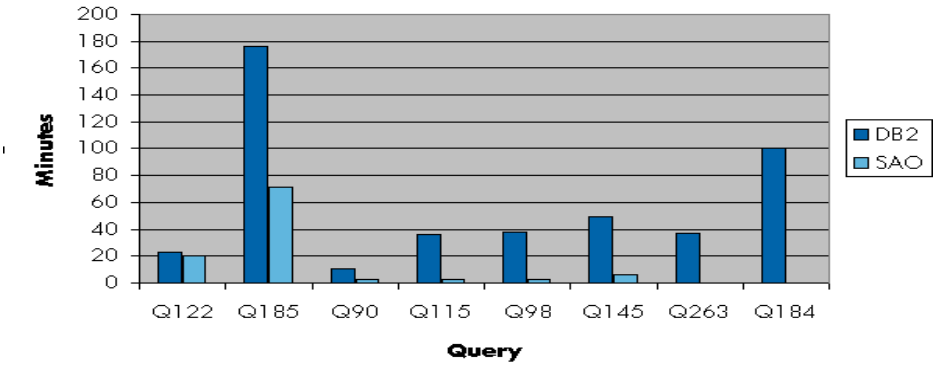


Fig. 8. Performance of ISAO relative to DB2 for z/OS on a beta customer workload

In case you thought we just got lucky in our choice of alpha customer workloads, Figure 8 is taken from a slide proudly shown by our beta customer at IBM's Information On Demand Conference (IOD) in October 2009, in which speed-ups ranged from 1.2 times to over 378 times! It's important to note that the speed-ups you enjoy with ISAO will depend greatly upon the schema, the data, and the query, as this chart graphically illustrates. Note that the y-axis in this graph is in minutes, not seconds, and that the long execution times for ISAO were measured early in the beta

and revealed some problems when dimension tables were bigger – relative to the fact table – than we'd assumed they would be. Once we addressed these issues, the speed-up ratios improved significantly, but the 378 times speed-up was on a different schema, in which the dimensions fit the original assumptions.

Skechers, a U.S. shoe retailer, uses Informix for their inventory and sales data warehouse having fact tables containing more than a billion rows. Queries took anywhere from a few minutes to 45 minutes to run on their production server running Informix 11.50. During the IWA Beta program, Skechers tested IWA with the same data and workload. The queries took just 2 to 4 seconds on the Informix Warehouse Accelerator, a 60x to 1400x speed-up, as shown in Table 1.

Table 1. Execution times for Skechers queries on Informix Warehouse Accelerator (with Informix 11.70) versus Informix 11.50

Query	Informix 11.50	IWA with Informix 11.70
1	22 mins	4 secs
2	1 min 3 secs	2 secs
3	3 mins 40 secs	2 secs
4	30 mins & up	4 secs
5	2 mins	2 secs
6	30 mins	2 secs
7	45 mins & up	2 secs

4 Blink Technology

Blink was designed from the ground up to exploit several important trends in hardware technology that are quite disruptive to database technology. Though Moore's Law continues to pretty much hold true, the increased density of chips is no longer resulting in ever-increasing clock rates, because the power consumption that that would require would result in power densities so great that chips would literally melt. So the industry has shifted toward using the increasing density of transistors on chips to have multiple CPUs[1], or cores, to achieve greater speed via parallel execution [BC 11]. Unlike most legacy software, Blink was designed explicitly to capitalize upon multi-core processors by chopping the data into small partitions, each of which can be independently processed by a core. As a main-memory database, we are of

[1] We may have to come up with a new acronym, since *Central* Processing Unit seems quite anachronistic in this multi-core world!

course exploiting the ever-increasing sizes, and decreasing costs, of commodity DRAM. And by spreading the data over multiple such processors and connecting them with a fast interconnect between the blades, we can accumulate quite large real memories in aggregate. Today's disk-based DBMSs were designed at a time when the latency gap between the time to randomly access something in main memory (DRAM) was only a few orders of magnitude from the time to access something on disk. As DRAMs have shrunk, so too has the time to access a particular byte in memory, while the time to move the disk arm to a particular piece of data on disk has not appreciably improved in decades. This means that there is an increasing gap of many orders of magnitude between DRAM and disk latencies, which propel us toward a main-memory DBMS. Hard disk drives are increasingly becoming archive storage!

4.1 What's So Disruptive?

The architecture of Blink exploits all these trends by starting with a clean slate, departing from three major tenets that database system designers have pretty much held sacrosanct for the last four decades. Given the extreme changes in hardware capabilities over the last four decades, it's actually amazing how well DBMS architectures have "weathered the storm", but we felt that the traditional architecture had been stretched to the breaking point, and it was time to rethink the whole architecture.

Tenet #1: Data warehouses are just too big to fit in main memory. The first major tenet is that one couldn't fit a useful amount of any data warehouse into main memory. So it was presumed that data warehouses had to be on disk, and disk dominated our thinking about costs, performance, data placement, and administration. Disk was the proverbial "800-pound gorilla" that could not be ignored, and we put all our energies into trying to minimize it, for good reason, as the gap between DRAM and disk arm latencies grew.

The disruption is that cheap main memories and, increasingly, flash memory, when aggregated over a large number of commodity machines, start to look pretty substantial, particularly when the data is compressed. As we calculated earlier, a Blade Center can hold over a terabyte of data in real memory, and that's using the most cost-effective memory DIMMs, not the largest ones possible (which are more expensive per byte). There seems to be no immediate barrier to this trend continuing. As a consequence, many data marts in today's enterprises, and some entire data warehouses of smaller enterprises, can begin to fit in a main-memory database. And when the disk bottleneck is removed, it is of course replaced with a new bottleneck, which seems to be the memory bandwidth, that is, the speed of moving things from DRAM into the cache, as well as the CPU speed itself. However, newer chips promise to have ever more cores, so probably memory bandwidth will increasingly be the new bottleneck. And since DRAM is byte-addressable, there is no preferred access path. In contrast, disk data is mapped to a linear address space, and moving to a new location involves an increasingly expensive (relatively) disk arm movement.

Tenet #2: The only way to scale to large databases is via indexes and materialized views. The second major tenet of databases, held for decades, was that the only way to scale up to really large databases was via indexes and pre-materializing frequently-accessed data in MQTs, cubes, etc. As a consequence, we needed an optimizer to choose among these various performance enhancers, and we needed a "4-star wizard" to choose which of the many such enhancers to create and maintain. To do that, we had to anticipate what queries might be posed in the future, usually using historical workloads that were assumed to be indicative of future workloads, a questionable assumption in BI, which is inherently *ad hoc*.

The disruption is that massive parallelism made possible by legions of commodity multi-core processors can divide and conquer large databases, so that scanning the data for every query becomes not only possible but desirable when accessing more than a few records. As a result, a user need only define what data is in the accelerator, rather than the dozens of materialized views and indexes that a standard database requires. We always scan the database for every query, although, as we'll see in a minute, entire portions of those scans can be eliminated when the literal in a predicate doesn't exist in the dictionary for that portion. Always performing the same query execution plan (a scan) significantly simplifies the accelerator, eliminating the need for this "performance layer" and an optimizer to choose among them, significantly reducing the total cost of ownership and the variance in response times.

Tenet #3: Existing disk-based DBMSs work fine for large memories – just define a big buffer pool! For many years, we believed that a main-memory database system was little different from a standard disk-based DBMS with a buffer pool large enough to obviate any disk I/Os. But what we have found in building Blink is that clever engineering can do far, far better. Examples of this are described in the rest of this paper, but include the exploitation of a proprietary compression technique that preserves the order of the underlying domain and uses fixed-length codes within database partitions, allowing us to perform most SQL operations on the encoded (compressed) values [RSQ+ 08]. This in turn allows Blink to fit multiple columns into a single register, so that predicates can be applied on all those columns simultaneously. We also exploit cache-conscious algorithms and special vector operations on multiple rows at a time. Lastly, we use hashing rather than sorting to perform grouping.

4.2 Data Layout and Compression

Figure 9 illustrates Blink's proprietary compression scheme, called frequency partitioning, which partitions each column's domain based upon the frequency of occurrence of values and creates a separate dictionary for each partition. For illustration purposes, consider a fact table of sales of products having various countries of origin, shown in the upper left-hand corner. The Blink loader independently histograms the values of each column, in this case the Origin and Product columns. It then partitions each histogram into partitions whose values are all encoded with the same fixed length. In our example, China and the USA are the

most common values, and need only one bit to represent those two values. Suppose the European Union countries are next most frequent; they can all be represented in 5 bits. And all the rest of the countries could be represented in perhaps 7 bits. Similarly, we can partition the Product column into the top 64 products, representable with 6 bits, and all the rest, perhaps requiring 11 bits. The intersection of these partitions defines what we call "cells" of all rows having one of the values in that partition and represented by a fixed-length code for each column. In our example, all the rows in Cell 1 have a country of origin of either China or the USA, represented in 1 bit, and one of the top 64 products, represented in 6 bits. But rows in Cell 6 represent countries other than China, the USA, or the European Union countries, in 7 bits, and an uncommon product, represented in 11 bits. Note that this scheme is an approximate Huffman scheme, in which the most frequent values are represented in the fewest number of bits [HRS+ 07]. What's important for our purposes is that field lengths are fixed within cells, but can vary from cell to cell. So queries must be recompiled for each cell to fit the lengths of that cell. And each cell has a dictionary that maps the values in that cell to the encoded values. By construction, encoded values are assigned in the same order as the original values of the domain in that cell, so that range predicates can also be applied on the encoded values. So in our example, China would be assigned the value 0, and the USA the value 1, according to alphabetic order.

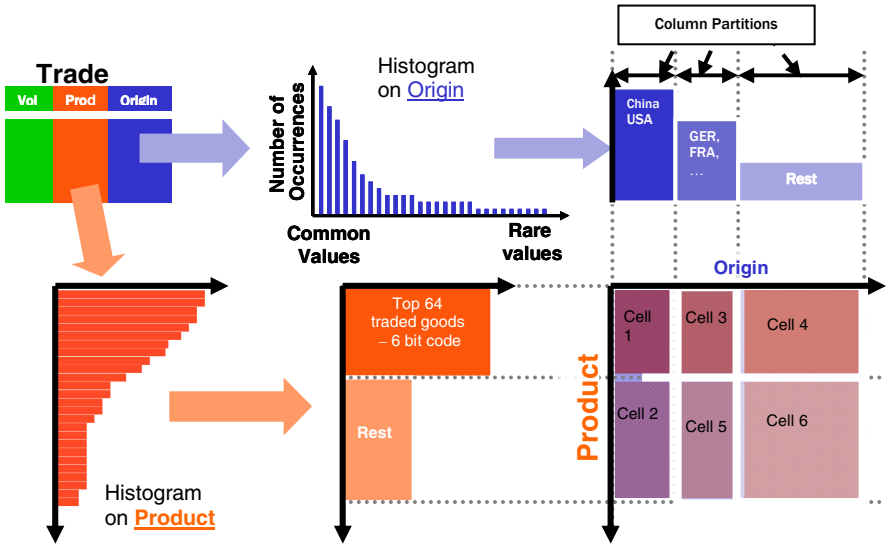


Fig. 9. Blink's Frequency Compression, applied to Origin and Product

Once the data has been partitioned with this scheme and dictionaries assign codes to each value, the Blink loader groups together all the encoded rows for a given cell into large, fixed-size (1MB) cell blocks. The number of rows that can fit in a block will depend upon the total length of each row when encoded, which depends upon which cell the rows are contained in. Blink uses the PAX format [ADH+ 01] for these

blocks, in which columns are grouped into column groups called banks, and all columns for a given set of rows are stored in the same block, as shown in Figure 10.

Blink stores columns in groups, or vertical partitions of the table, called "banks". The fraction of a row (or tuple) that fits in each bank is called a "tuplet". The assignment of columns to banks is cell-specific, because the column lengths vary from cell to cell. The assignment uses a bin-packing algorithm that is based upon whether the column fits in a bank, whose width is some fraction of a word, rather than usage in a workload, but access patterns could also be considered [BSS 11]. In this way, multiple tuplets can be assembled in a register side-by-side without having to shift bits, so that multi-row parallelism is also achieved in SIMD (single-instruction, multiple data) fashion. Also, scans need only access the banks that contain columns referenced in any given query, saving scanning those banks with no columns referenced in the query. This projection is similar to the way pure column stores avoid disk I/Os to great advantage. The savings in Blink aren't as dramatic because there are no disk I/Os with Blink (remember, it's a main-memory database!), but it still saves considerable CPU.

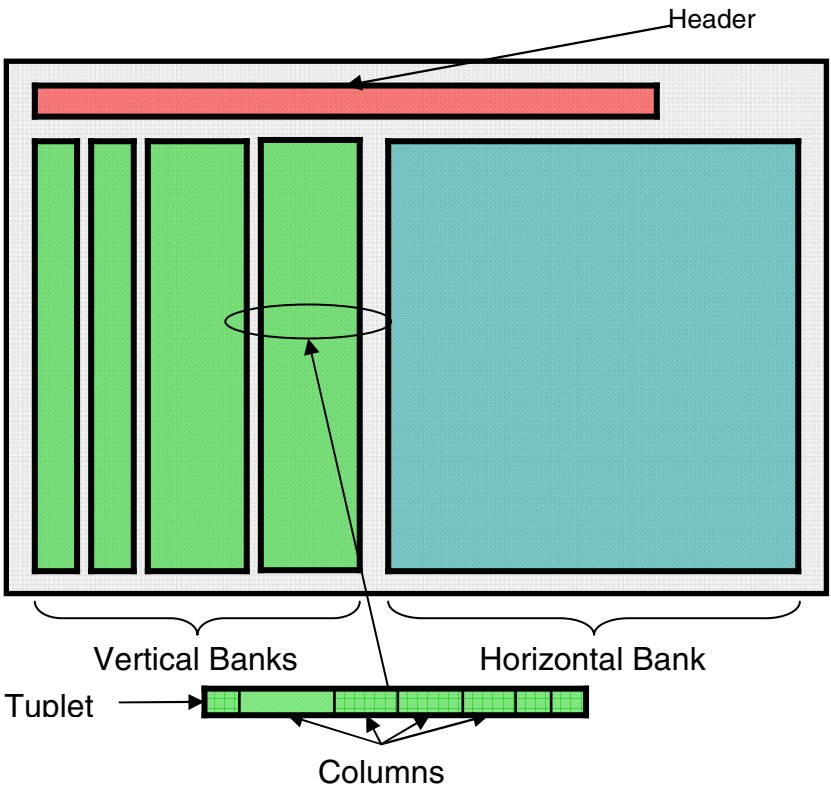


Fig. 10. Storage layout used by Blink blocks of 1 MB

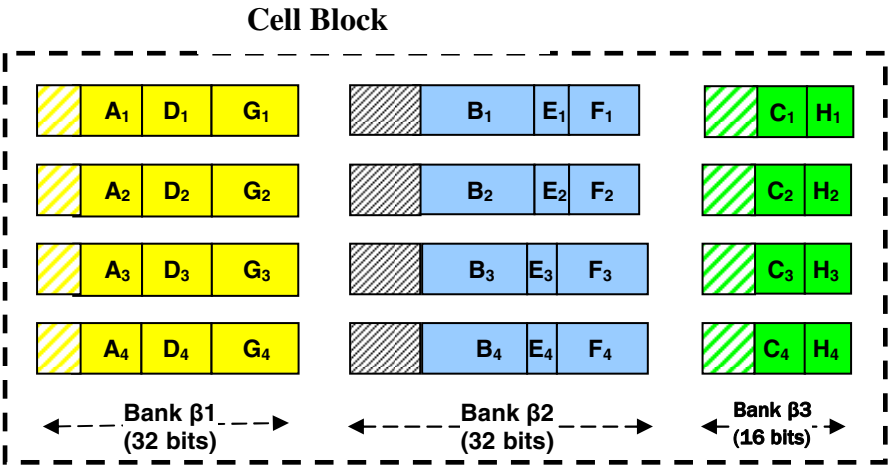


Fig. 11. Banks in a cell block in Blink's page format

In the example shown in Figure 11, columns A, D, and G are grouped into one 32-bit bank, β1, while B, E, and F are grouped into another 32-bit bank, β2, and C and H are grouped into a 16-bit bank, β3. Though shown row-wise in this figure for expository purposes, the tuplets in fact are padded if necessary to exactly fit the bank width and stored together continuously as a vector. In this way, several tuplets can be loaded into a register for processing without having to do any bit shifting. In this example, 4 tuplets from bank β1 can be loaded at a time into a register, as shown in Figure 12 below, and processed simultaneously in a SIMD fashion, as described in the next section.

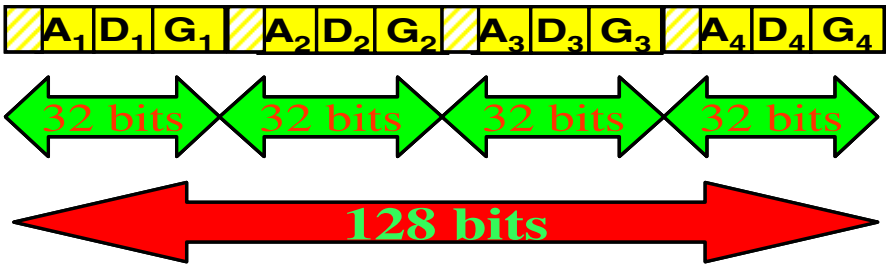


Fig. 12. Multiple tuplets from a bank loaded into a 128-bit register for SIMD operations

4.3 Query Execution

The scheme we discussed above for encoding data is also the scheme for dividing the data finely into chunks for processing. The cell is not only the unit of encoding, but also the unit of processing. Each qualifying cell is assigned to one core, and has its own hash table in cache, as shown in Figure 14. This ensures that each core can process its data with no shared objects, obviating the need for any locking or latching

or mutexes. Processing of a cell is all done in the context of a scan over all rows in that cell, applying predicates, hashing the GROUP BY column(s) to group, and then performing the aggregate functions. These operations can all be performed on the encoded values; only arithmetic operations (e.g., in predicates and aggregate functions) require that data be decoded. This means that each cell can operate completely independently on its piece of the database, and the response time is therefore roughly proportional to the database size divided by the aggregate number of cores across all nodes – "embarrassingly parallel".

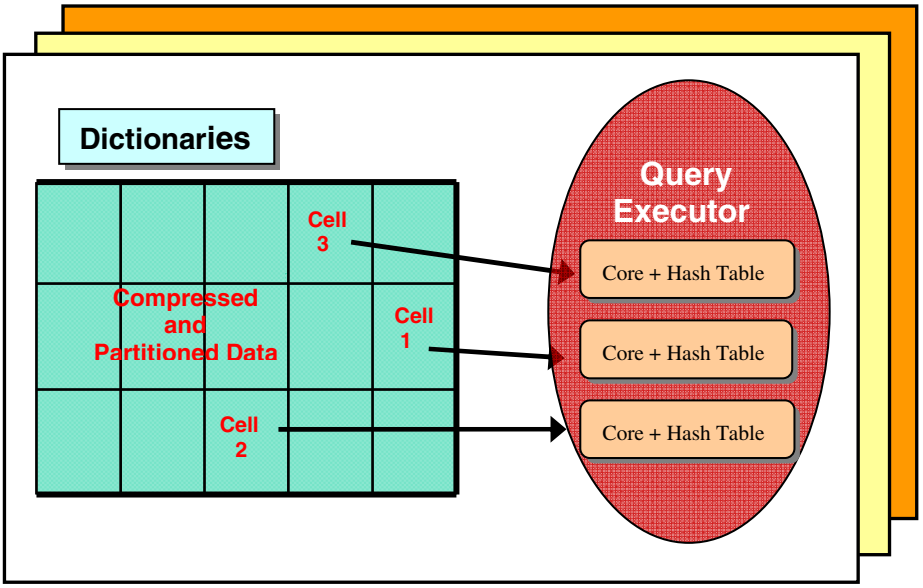


Fig. 13. Each compressed cell is dynamically assigned to a core (thread), which has its own hash table

Looking a little closer at how this processing happens, when a query begins execution, all the cells for the table being scanned are put in a work queue. Each cell in the queue is assigned to a thread running in a core. As shown in Figure 14, first to be executed are so-called "fast-path" predicates, which include equality, range, and short IN-list predicates, which can be performed in just a few instructions, as we shall soon see. Any other predicate, especially a predicate containing expressions or a join predicate, is called a "residual" predicate[2]. Processing residual predicates requires decoding values, except for join predicates. Next the GROUP BY columns are hashed to find the group, and lastly aggregate functions are calculated. The color scheme in Figure 14 shows that most operations can be performed on the encoded values, denoted by green, but any arithmetic operations require decoding, shown in beige. When all rows have been so processed for a cell, the hash tables and aggregate functions for each thread are merged on a single (worker) node, and the resulting

[2] This term should not be confused with a similar term in IMS and DB2.

merged hash table is sent to the coordinator, where the results of all workers are finally merged, and final processing, such as ORDER BY, is performed before returning the results to DB2 and the user.

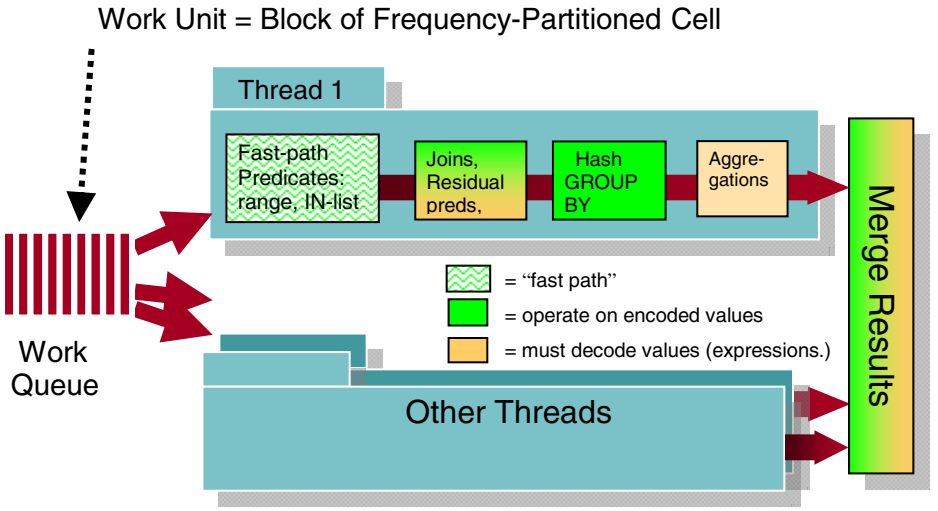


Fig. 14. Each thread independently performs all operations on its cell. Then all threads merge their results.

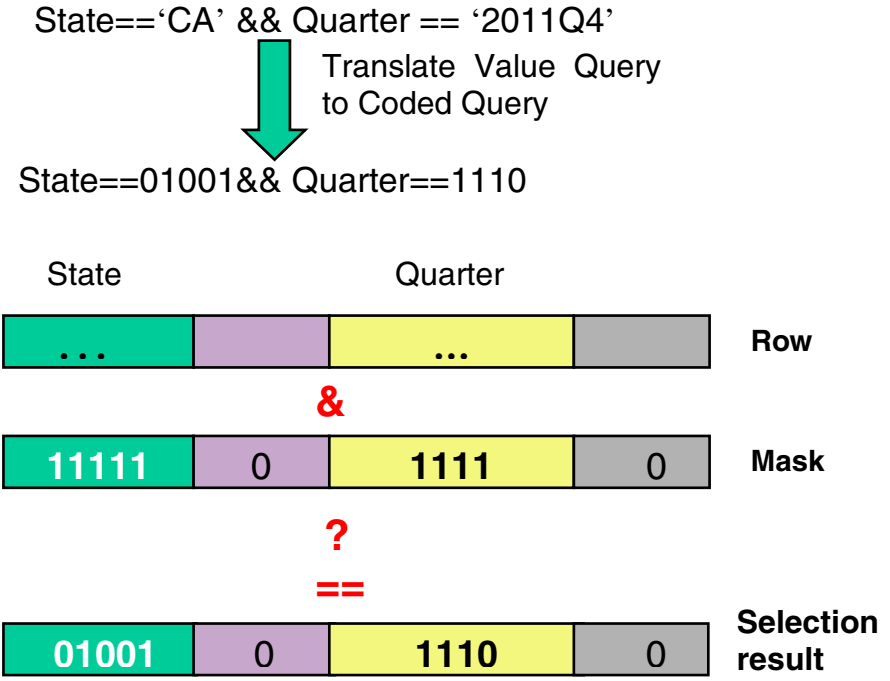


Fig. 15. Applying predicates to multiple, encoded columns simultaneously

Fast-path predicates can be applied simultaneously to multiple columns that remain encoded, so many of them can fit into a 128-bit register, as shown in Figure 12. At compile time, the literals are encoded by looking them up in the dictionary and converting the literal to its encoded value. In Figure 15, for example, the literal 'CA' is found in the dictionary for the state column, and is converted to '01001'. Similarly, 'Q4' is encoded into '1110' using the dictionary for the Quarter column. If the value cannot be located in the dictionary for that cell, then that value never occurs in that cell, by construction, and so we can eliminate all rows in that cell from consideration at compile time. A mask is also constructed that masks out the columns not having predicates. Then, at execution time, each row is ANDed with the mask, and compared to the encoded literal. In this way, as many predicates as columns fit into a register can be applied simultaneously. This is much faster than how a traditional DBMS applies predicates to columns, loading in succession each (un-encoded) column value and comparing it to its (un-encoded) literal. A variant of this technique for fast-path predicate evaluation can even be used for applying multiple *range* predicates simultaneously, as shown in [JRS+ 08].

Since Blink is good at doing scans, joins are performed by re-writing each join into a succession of scans. In the first step, Steps 1(a) and 1(b) in Figure 16, each dimension is scanned, applying predicates local to that table. Qualifying rows then add their join-column values (usually a primary key[3]) to an IN-list (actually a hash table) that will be used in the fact-table scan, and their "auxiliary columns" (columns used later in the query) are inserted into a hash table. This is repeated for each dimension. The fact table is then scanned, as shown in step (2) in Figure 16, where the IN-lists from each dimension are used to filter the corresponding join columns (usually a foreign key[12]) of the fact table, along with any predicates local to the fact table (shown as σ_f). Then the auxiliary columns are fetched from the hash table, and used to do grouping and aggregation. Note that the join columns of the fact and dimension tables, though drawn from the same domain, are encoded independently, so in reality the join column values from each dimension must be decoded and re-encoded with the dictionary of the corresponding join columns in the fact table before being added to the IN-list. For schemas having more than one level of join, this technique is applied recursively, starting from the outermost dimension tables and joining inward. In such cases, a table that is neither the outer-most nor the central fact table can act as both a "fact" table and as a "dimension" table in successive joins.

Grouping exploits the same hash-table technique that is used for joins. Rows from the scan of the fact table that qualify the probes of the hash tables for each dimension are subsequently hashed and inserted into the grouping hash table. The payload for each hash bucket is the set of aggregate functions for that query. Since each thread has its own hash table, the final result is computed by merging these into a global hash table, from which the result rows are output.

[3] Note that, although the join columns for each dimension usually form a primary key (PK), and the join columns for the fact table usually define a foreign key (FK), this is not required for Blink joins, nor is any referential integrity.

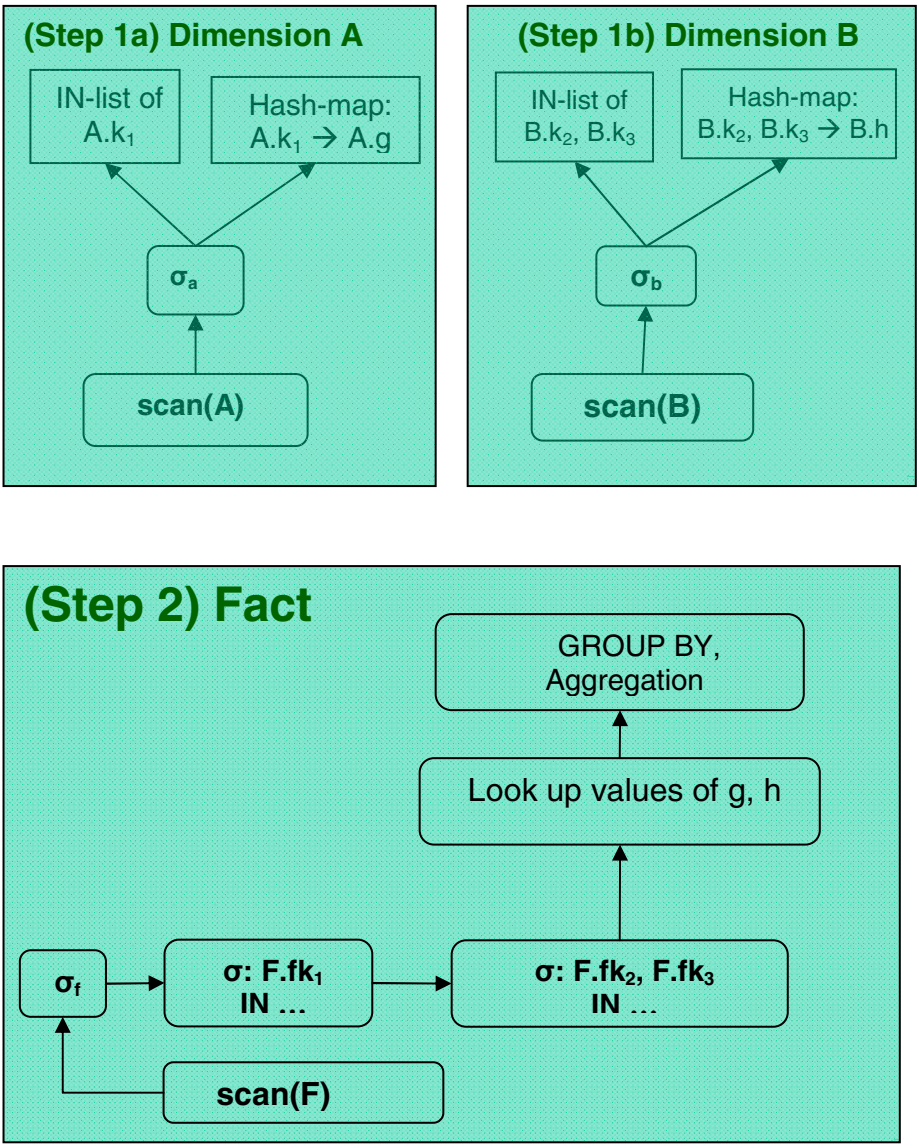


Fig. 16. Joins are done as a sequence of scans of the dimension table, and then the fact table

5 Related Work

Fast processing of Business Intelligence queries is currently a "hot topic" in both academia and industry, with numerous new systems in academia and start-ups that are rapidly being bought up by established players. Unfortunately, most of the

commercial systems have offered little or no documentation of their systems in the refereed literature, preferring instead "white papers" for marketing purposes, so it is often difficult to discern precisely how they store data and process queries.

Vertica [Vert 11] is the product version of the academic project C-store [SAB+ 05]. C-store exploits bulk processing, late tuple reconstruction, and compression. In addition, C-store introduced the concept of projections as indexes in a column-oriented DBMS. A projection is a copy of (part of) a table, sorted based on a leading column; it enables fast selections and minimizes tuple reconstruction costs, while exploiting run-length encoding compression to minimize the storage overhead. In addition to the above features, Vertica also introduced FlexStore, a feature that allows for hybrid storage, in which multiple columns are grouped together so that they can be accessed and updated faster.

VectorWise [Vect 11] is the product version of the academic project MonetDB/X100 [BZN 05]. VectorWise introduced the vectorized execution model, in which data is processed one chunk of a column at a time, as opposed to one tuple at a time in pure row-stores or one column at a time in pure column-stores. This allows for better memory utilization and scalability due to smaller intermediate results, compared to a pure column-store. Furthermore, VectorWise introduced lightweight compression schemes that improved not only I/O but the complete process of accessing data from disk and decompressing it. In addition to columnar storage, VectorWise also exploits the PAX storage scheme, which stores together columns that may benefit from simultaneous access in queries.

Hyrise is a recent academic project that exploits hybrid storage in a main-memory database system for mixed (OLTP and BI) workloads [GKP+ 10]. Its main feature is an offline analysis tool that, given a workload and a database schema, can decide the proper physical layout to optimize performance. Hyrise includes a detailed cost model for each database operator, mainly to model cache misses, the primary cost in a main-memory environment. Its main decision is which columns should be stored together and in what combinations when first loading it. Columns used for analytical queries are stored alone, in columnar fashion, while columns used for OLTP-like queries are stored together in the form of rows.

SAP's Netweaver Business Warehouse Accelerator (BWA) is a main-memory column store database system that uses dictionary encoding to pack multiple values in a register, and exploits SIMD operations to do decompression and (local) predicate evaluation, as does Blink. However, unlike Blink, the values of only one column are packed without padding to align to register boundaries, so that values may span register boundaries, creating challenges for processing values on those boundaries and extracting results using a clever series of complicated bit shifts [WPB+ 09].

Hyper is another recent academic project aiming to bridge the gap between OLAP and OLTP systems [KN 11]. It is a main-memory database system in which data may be stored either as columns or as rows. The main feature of Hyper is that it exploits the ability of modern hardware and operating systems to create virtual memory snapshots for duplicated processes. Pages are duplicated on demand when OLAP queries conflict with OLTP queries. This allows OLAP queries to see a very recent snapshot of the data, while OLTP queries can continue in parallel.

Compared to existing systems such as Vertica and VectorWise, Blink is closer to VectorWise, as it also processes chunks of a table at a time. Blink introduces a more advanced compression scheme, frequency partitioning, which allows it to achieve a good balance between reducing size while still maintaining fixed-width arrays. This drastically changes the way data is stored and accessed compared to a pure column store. Hyrise and Hyper are interesting main-memory technologies, but their real-world feasibility remains to be proven, as they are very recent efforts. In particular, the results from Hyrise may be valuable for organizing banks in Blink offline.

6 Conclusions

In conclusion, radical changes in hardware necessitate radical changes in software architecture. Blink is such a radically novel architecture – a main-memory, special-purpose accelerator for SQL querying of BI data marts that exploits these hardware trends. It also exploits proprietary order-preserving compression techniques that permit SQL query processing on the compressed values, and evaluating multiple predicates on multiple columns simultaneously, using cache-conscious algorithms. As a result, Blink can process queries in simple scans that achieve near-uniform execution times, thus speeding up the most problematic queries the most, without requiring expensive indexes, materialized views, or tuning. This radical simplification by completely obviating the need for a tunable "performance layer" is the best way to lower administration costs, and hence the total cost of ownership.

References

[ADH+ 01] Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. In: VLDB 2001, pp. 169–180 (2001)

[BC 11] Borkar, S., Chien, A.A.: The Future of Microprocessors. Comm. of the ACM 54(5), 67–77 (2011)

[BSS 11] Beier, F., Stolze, K., Sattler, K.-U.: Autonomous Workload-driven Reorganization of MMDBS Data Structures. In: 6th International Workshop on Self-Managing Data Bases (SMDB 2011), Hanover, Germany (2011)

[BZN 05] Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-Pipelining Query Execution. In: CIDR 2005, pp. 225–237 (2005)

[GKP+ 10] Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudré-Mauroux, P., Madden, S.: HYRISE - A Main Memory Hybrid Storage Engine. PVLDB 4(2), 105–116 (2010)

[HRS+ 07] Holloway, A.L., Raman, V., Swart, G., DeWitt, D.J.: How to barter bits for chronons: compression and bandwidth trade offs for database scans. In: SIGMOD 2007, pp. 389–400 (2007)

[IBM 10] IBM Corp., IBM Smart Analytics Optimizer for DB2 for z/OS V1.1 User's Guide, IBM Corp., Tech. Rep. (November 2010)

[JRS+ 08] Johnson, R., Raman, V., Sidle, R., Swart, G.: Row-Wise Parallel Predicate Evaluation. In: VLDB 2008, pp. 622–634 (2008)

[KN 11] Kemper, A., Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE 2011, pp. 195–206 (2011)

[QRR+ 08] Qiao, L., Raman, V., Reiss, F., Haas, P., Lohman, G.: Main-Memory Scan Sharing for Multi-core CPUs. In: VLDB 2008 (2008)

[RS 06] Raman, V., Swart, G.: How to wring a table Dry: Entropy Compression of Relations and Querying Compressed Relations. In: VLDB 2006 (2006)

[RSQ+ 08] Raman, V., Swart, G., Qiao, L., Reiss, F., Dialani, V., Kossmann, D., Narang, I., Sidle, R.: Constant-time Query Processing. In: ICDE 2008, pp. 60–69 (2008)

[SAB+ 05] Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., Zdonik, S.B.: C-Store: A Column-oriented DBMS. In: VLDB 2005, pp. 553–564 (2005)

[SBS+ 11] Stolze, K., Beier, F., Sattler, K.-U., Sprenger, S., Grolimund, C.C., Czech, M.: Architecture of a Highly Scalable Data Warehouse Appliance Integrated to Mainframe Database Systems. In: Database Systems for Business, Technology, and the Web (BTW 2011) (2011)

[SH 02] Schmuck, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: FAST 2002: Proceedings of the 1st USENIX Conference on File and Storage Technologies, pp. 19–29. USENIX Association, Berkeley (2002)

[Vect 11] http://www.sigmod.org/publications/sigmod-record/1109/pdfs/08.industry.inkster.pdf

[Vert 11] http://www.vertica.com/wpcontent/uploads/2011/01/VerticaArchitectureWhitePaper.pdf

[WPB+ 09] Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. In: PVLDB 2009, pp. 385–394 (2009)

MemcacheSQL
A Scale-Out SQL Cache Engine

Qiming Chen, Meichun Hsu, and Ren Wu

HP Labs
Palo Alto, California, USA
Hewlett Packard Co.
{qiming.chen,meichun.hsu,ren.wu}@hp.com

Abstract. Real-time enterprise applications and data-intensive analytics often require very low latency access to large volumes of data. In-memory data cache is a promising approach to enable real-time analytics. In this paper we examine the issue of scaling out memory cache over multiple machines while providing a common data query language with rich expressive power, and allowing the data cached in memory to persist with the ACID properties. Designing and building a product quality scaled-out data cache engine from scratch is an option, but it requires large investment in time and engineering efforts.

We propose to build such a system by extending existing SQL systems. In our approach, we extend database buffer pool with a Distributed Caching Platform (DCP). We have developed a prototype of the system, which we call MemcacheSQL, engine by integrating PostgreSQL's buffer management with Memcached, a widely used DCP platform. Our contributions include optimized data coherence management along the memory hierarchy, and flexibly configurable page buffering policies for accommodating various workloads. Our preliminary experiments show that the proposed system has the potential to achieve scalable performance gains by dynamically leveraging scale-out memory nodes, while retaining the full SQL's expressive power and DBMS's ACID support.

1 Introduction

Many data analysis applications are designed to store and exchange intermediate or final results through database access, which very often becomes their performance bottleneck. With increased memory bandwidth, there are two general solutions to the above problem: one is to modify the data analysis algorithms and programs to take advantage the availability of larger memory space; the other is to provide in-memory data stores to accelerate the required data access. The former would require rewriting many programs, and the latter would provide a common basis for enhancing the performance of these programs without re-writing them. This has motivated us to develop a data cache engine with SQL interface to handle memory cache distributed over multiple machine nodes.

M. Castellanos, U. Dayal, and W. Lehner (Eds.): BIRTE 2011, LNBIP 126, pp. 23–37, 2012.
© Springer-Verlag Berlin Heidelberg 2012

1.1 The Issues

In developing MemcacheSQL, we face the following issues.

First, the memory resource on a single server is limited and used by several system components such as file buffer, database buffer, applications, etc. The key to scaling-out database buffer pool is to provide a unified cache view over multiple server nodes, referred to as memory nodes. For this purpose we chose to adopt the widely accepted Distributed Caching Platform (DCP) [1,6-11] – Memcached [6].

Next, a common data retrieval language with rich expressive power is necessary for accommodating various applications. For this requirement we believe SQL is a reasonable choice.

Further, the memory-resident data must be backed up by a persistent store, to guard against system shutdown or failure, while maintaining adequate correctness semantics.

1.2 Our Solution

We tackle the above problems by extending the database buffer pool with DCP running on multiple memory nodes with high-speed inter-connections. This approach allows us to leverage the full SQL expressive power, the ACID properties of transaction management, as well as the scalability and availability of NoSQL distributed Key-Value storage technologies.

Our architecture provides a unified cache view that potentially allows a database engine to access buffers across multiple machines. With such architecture the eviction strategy of the database buffer pool and the DCP buffer pool as well as the data transfer between the two have to be addressed. With two buffer pools, the consistency of the contents must be managed since these two buffer pools reside on individual servers and may use their own page replacement algorithms (e.g. LRU (Least Recently Used), MRU (Most Recently Used), etc.). In addition, how to minimize the data transfer between the two, and how to ensure consistent data access in the presence of multiple distributed data stores, are key to system performance and reliability.

We have built a prototype of the proposed system, which we call MemcacheSQL, by integrating PostgreSQL's buffer management with Memcached. Our contributions include optimized data coherence management along the memory hierarchy, and flexibly configurable page eviction policies for various workloads. Since we have all the PostgreSQL DBMS functionalities retained, the system is robust against failures of any of the connected Memcached server.

MemcacheSQL offers memory based data analytics by taking advantage of scale-out memory nodes, enabling data-intensive analytics application to potentially run much more efficiently. Integrating the query engine's buffer management with DCP leverages both the mature DBMS capabilities as well as the memory scale-out capabilities of the DCP technology.

1.3 Related Work

Several database products such as Oracle, MySQL, EnterpriseDB are provided with DCP interface for database applications such as those coded in PL-SQL, PSQL, or

User Defined Functions (UDFs), to take advantage of DCP's unified cache view [1,6-11]. While these represent effective uses of DCP for storing application data, they are are orthogonal to extending database buffer pool with DCP.

There have not been many efforts in using DCP to extend database buffer pool. One project has been reported by the Waffle Grid Project on MySQL database [13,14], which is characterized by "overflowing" the evicted data pages from the buffer pool to the caches located on other machines which are managed under Memcached. We share the same motivation but with a different architecture. The problem of such "overflow" approach is that with big data, the page swapping between the buffer pool and the cache managed by Memcached is too frequent; and such data transfer overhead would diminish the benefits of using distributed caches. Instead, we provide the memory hierarchy architecture to reduce such data swapping but still enforce consistent data access. We further support flexibly configurable page eviction policies for accommodating various workloads. We also take into account the stream data captured on the fly by queries [4] rather than read from disks. All these constitute to our new contributions.

The proposed MemcachSQL is also different from Membase [15], a storage engine for Memcached's key-value pairs. Our MemcacheSQL is characterized by the full SQL interface and DBMS functionality, and is implemented by extending the database buffer pool management directly.

The rest of this paper is organized as follows: Section 2 provides overviews to database buffer management and DCP; Section 3 describes how to scale out buffer pool with DCP, and compares the overflow model and the inclusion model for integrating buffer pool and DCP's memory space; Section 4 shows our preliminary experimental results; Section 5 concludes the paper.

2 Database Buffer Pool and Memcached

A primary goal of our research is to provide the scalable cache engine but leveraging SQL's rich expressive power and query engine's mature data management capability, which is very different from building an in-memory data store from scratch.

2.1 Database Buffer Management

We shall refer to PostgreSQL database engine to describe our solution. In the PostgreSQL database, each table is physically stored in file system under a subdirectory. In that directory there are a number of files. A single file holds certain amount, up to 1GB of data. The file is treated as a series of blocks (i.e. pages) each block holds 8K bytes.

A database buffer pool is a memory based data structure - a simple array of blocks, also called pages, with each page entry pointing to a binary memory of certain size. For this paper, we have chosen to fix the page size at 8K. A page in the buffer pool is used to buffer a block of data in the corresponding file, and identified by the table space ID, table ID, file ID and the sequence number of the block in the file.

The buffer pool is accompanied by a buffer pool descriptor that is also an array of data structures called buffer descriptors. Each buffer descriptor records the

information about one page, such as its tag (the above table space, table, file IDs and the block number), the usage frequency, the last access time, whether the data is dirty, etc. Maintaining the buffer pool allows the pages to be efficiently accessed in memory without being retrieved from disks.

When a query process wants a page corresponding to a specific file/block it needs, if the block is already cached in the buffer pool, the corresponding buffered page gets pinned; otherwise, a new page must be found to hold this data. If there are no pages free (there usually aren't), the process selects a page to evict to make space for the new one. If the selected old page is dirty, it is written out to disk asynchronously. Then the block on disk is read into the page in memory.

Pages all start out pinned until the process that requested the data releases (unpins) them. The pinning process, whether the page was found initially or it had to be read from disk, also increases the usage count of the page. Deciding which page should be removed from the buffer pool to allocate a new one is a classic computer science problem. The usual strategy is based on Least Recently Used (LRU): always evict the page that has gone the longest since it was last used. The timestamp when each page was last used is kept in the corresponding buffer descriptor in order for the system to determine the LRU page; another way keeps pages sorted in order of recent access. There exist other page eviction strategies such as Clock-Sweep, Usage-Count, … etc. For simplicity, hereafter we generally refer to the page to be evicted as the LRU page.

With limited memory space in the buffer pool, much of the data has to be handled through file access. However, as shown in in Fig 1, the file system also supports in-memory buffering, and the file buffer and the database buffer compete for memory resources. In general every page contained in the database buffer pool is also contained in the file buffer. Therefore, as a database configuration principle, a buffer pool should be allocated with moderate memory space.

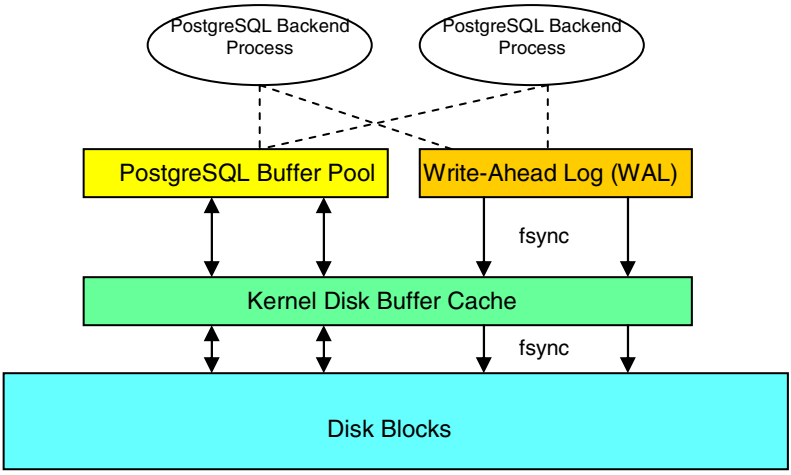


Fig. 1. Database buffer and file buffer

2.2 Distributed Cache Platform (DCP)

A DCP provides the unified cache view over multiple machine nodes, which allows multiple processes to access and update shared data [6-11]. As illustrated in Fig. 2, a DCP virtualizes the memories on multiple servers as an integrated memory; it provides simple APIs for key-value based data caching and accessing, such as *get(), put*(), *delete*(), etc, where keys and values are objects.

Memcached [6] is a general-purpose distributed memory caching system that provides a giant hash table distributed across multiple machines, or memory nodes. The data are hash partitioned to these memory nodes. When the hash table on a node is full, subsequent insert causes older data to be purged in Least Recent Used (LRU) order.

Memchached uses the client–server architecture. The servers maintain a key–value associative array; the clients populate this array and query it. Keys are up to 250 bytes long and values can be at most 1 megabyte in size.

Clients use client side libraries to contact the servers which, by default, expose their service at a specific port. Each client knows all servers; the servers do not communicate with each other. If a client wishes to set or read the value corresponding to a certain key, the client's library first computes a hash of the key to determine the server that will be used. Then it contacts that server. The server will compute a second hash of the key to determine where to store or read the corresponding value.

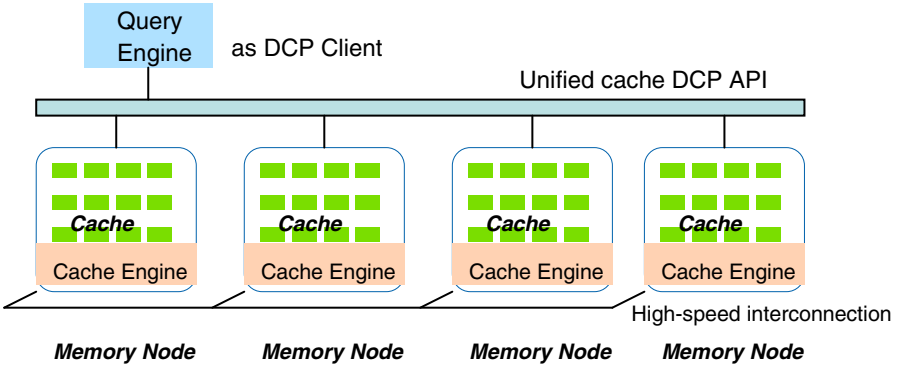


Fig. 2. DCP **servers p**rovide to a query engine the unified and scalable cache across multiple DCP memory nodes

2.3 Mapping Buffer Blocks (Pages) to Key-Value Pairs

A database buffer pool is an array of data blocks or pages, and for PostgreSQL the page size is 8KB. Each page can be treated as a binary object.

Memcached supports a unified key-value store across multiple server nodes. On each node an individual Memcached server is launched for managing the local portion of the key-value store.

In integrating PostgreSQL query engine with Memcached based DCP infrastructure, the query engine acts as the DCP client that connects to the Memcached server pool including multiple distributed Memcached servers. These servers cooperate in managing a unified in-memory hash table across multiple nodes. The basic idea of extending the buffer pool with Memcached is based on storing buffered pages in such a unified hash table as key-value pairs, where the pages are hash partitioned to separate portions of the unified hash table residing on separate nodes.

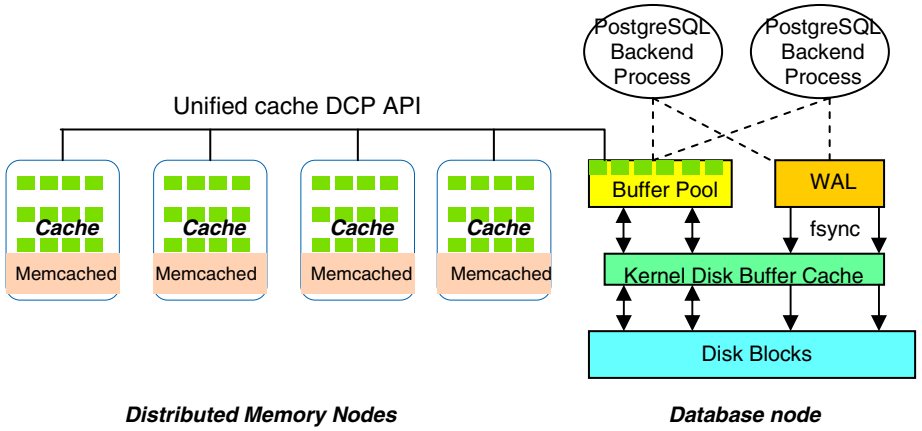


Fig. 3. Extend PostgreSQL shared buffer pool to DCP

The mapping of a buffered page to a key-value is handled in the following way.

- **Key.** The identifier of the page (i.e. buffer block), composed of the table space ID, table ID, file ID and the series number of the block in the file, is serialized to a string key.
- **Value.** The 8KB binary content of the page is treated as the value corresponding to the page key. Note that the Memcached's Binary Protocol is adopted such that a value is passed to the API functions of data transfer by the entry pointer plus the length of bytes.

3 Scale-Out Buffer Pool with Memcached

At the conceptual level, the MemcacheSQL combines the advantages of SQL's expressive power, DBMS's ACID support, as well as NoSQL distributed key-value store's scalability and availability.

We extend the buffer manager of the PostgreSQL engine to allow buffered pages to be moved to and retrieved from Memcached, the DCP infrastructure. We assume in MemcacheSQL, the size of memory available in Memcached is always larger than that available in the database buffer. The query engine acts as a DCP client. The buffered pages may be sent to different physical Memcached sites based on the hash value of the page key. As a result, the buffered pages are placed in multiple "memory nodes" but can be accessed with the unified interface. The following basic mechanisms are adopted in architecting our system.

- We treat the DCP cache as additional buffer space for the database buffer pool, with all the concurrency control, page eviction management and file I/O still handled under the database buffer pool manager. Any page to be cached in Memcached always goes through the buffer pool manager; any page retrieved from Memcached also goes through the buffer pool under the control of the buffer pool manager. There are no file I/O and database buffer management functionalities such as page locking, on the Memcached sites. This greatly simplifies the overall system design and ensures data consistency.
- We force the dirty pages to be kept in the buffer pool, and when they are to be evicted from the database buffer pool, if the pages exist in Memcached, then the corresponding pages held in Memcached are updated to reflect the current values. Therefore if a page does not exist in the buffer pool but exists in Memcached, then its content is up to date in Memcached. Note that the inverse may not be true. This way, the Memcached cache never has "out of date" data when accessed. Note also that the database buffer pool is shared by all the running queries, therefore the dirty page caused by updates from any query is centrally handled by the buffer pool manager.
- The DCP memory nodes are private to the query engine. This ensures no other programs can alter the content of the buffered pages. This restriction can be relaxed to allow multiple query executors to connect to the same storage engine; however this aspect is not addressed in this paper.

Below we discuss the algorithms corresponding to two page buffering models.

3.1 The Overflow Model of Page Buffering

Under the overflow model of page buffering, given the buffer pool B and the DCP buffering space D (physically located in distributed memory nodes), the unified page buffer is B $\cup$ D, and B $\cap$ D = empty. A page evicted from B is moved to D. Any page p, although can be moved between B and D, can only be pinned when $p \in$ B.

As illustrated in Fig. 4(a), under this model, when a process requests a page (8KB block), it first tries to get the page from the local buffer pool; if thye page is not found then it tries to get the page from DCP; if the page is not in DCP then the page is loaded from disk. In case the buffer pool is full, it chooses a page to be evicted based on certain replacement policy (e.g. LRU, MRU, etc). In our current design, we have chosen to use LRU as the replacement policy. The chosen page is evicted and moved to DCP. Each DCP (Memcached) site also enforces its own replacement policy.

More specifically, when a process wants a page, the following buffer allocation algorithm is used.

- If the page is already in the DB buffer, it gets pinned and then returned; pinning a page prevents the page from being selected for eviction before it is safe to do so, since the buffer pool is shared by all the backend processes;
- Otherwise, a new buffer block must be found to hold this page. If there is a free block, then it is selected and pinned; else the LRU page is selected to be evicted to make space for the new one; the evicted LRU page is transmitted to Memcached to be cached; further, if the LRU page is dirty (updated since loaded), it is written out to disk;

– With a slot in the buffer pool available, the system first tries to load the page content from DCP if the page is found in DCP; otherwise the new data is loaded from disk.

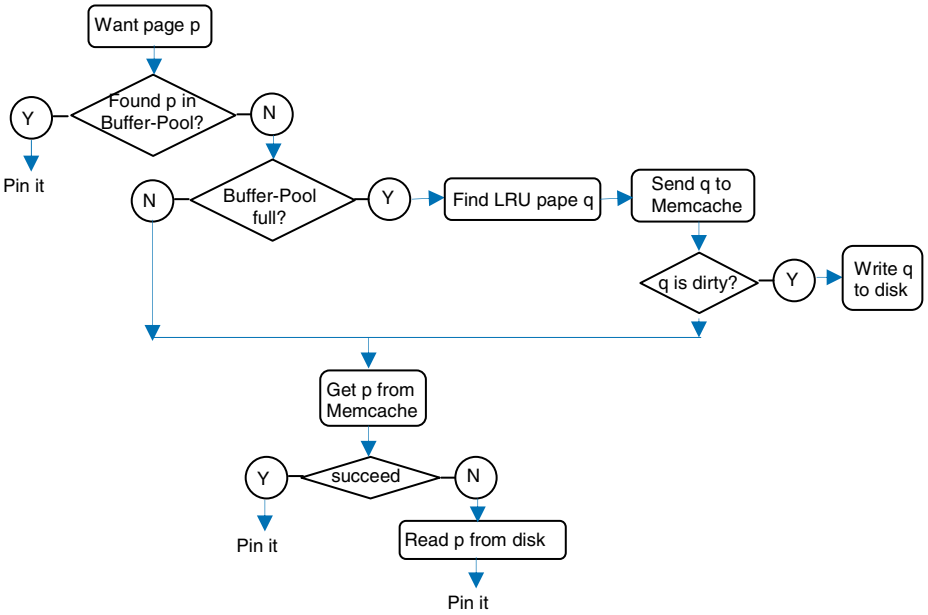


Fig.4(a). Integrate buffer management with DCP under the *overflow* model

With this algorithm, all the evicted pages, dirty or not, are written to Memcached space, possibly overwriting the existing pages in Memcached with the same page keys. Conceptually the Memcached space provides an overflow buffer for the pages cached in the buffer pool. All the system control and data control functions are performed by the database buffer manager with the ACID properties retained.

3.2 The Inclusion Page Buffering Model

The inclusion model is not a new concept; it has been proposed and studied in the context of processor memory hierarchy for analyzing cache coherency policies (e.g. [16]). Its implication for DCP-extended database cache is the purpose of our study. Under the inclusion page buffering model, given the buffer pool B and the DCP buffering space D (physically located in distributed memory nodes), the unified page buffer is B $\cup$ D, and B $\subseteq$ D.

As illustrated in Fig. 4(b), under this model, when a process requests a page (8KB block), the system first tries to get the page from the local buffer pool; if the page is not found then it tries to get the page from DCP; if the page is not in DCP, then the page is loaded from disk and copied it to DCP. In case the buffer pool is full, the LRU page is evicted, and if the page is dirty, it is also transmitted to DCP to refresh DCP's copy.

More specifically, when a process wants a page, the buffer allocation algorithm works in the following way:

- If the page is already in the DB buffer, it gets pinned and then returned; pinning a page means lock it or wait for getting the lock since the buffer pool is shared by all the backend processes;
- Otherwise, a free block is checked, if found then it is pinned for using, else the LRU page is selected to evict to make space for the new one; the evicted LRU page is *transmitted to DCP only if it is dirty* (updated since loaded); in that case, it is also written out to disk through another asynchronous process;
- With a slot in the buffer pool available, the system first tries to load the page content from DCP if it is found in DCP; otherwise the data is loaded from disk, as well as *copied to the Memcached space*.

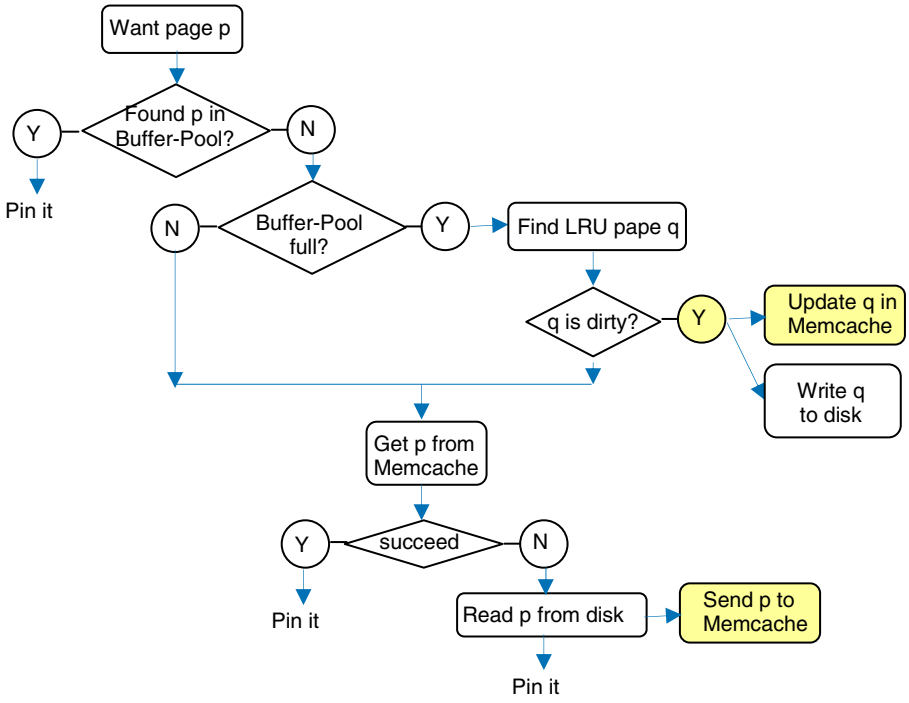


Fig. 4. (b). Integrate buffer management with DCP under the *inclusion* model

With this algorithm, all the pages read from disk are copied to the Memcached space. However, only the evicted pages which are dirty are written to Memcached. Conceptually the pages cached in the buffer pool are included in the Memcached space.

3.3 Switchable Policies of Integrated Page Buffering

There is a tradeoff between the overflow model and the inclusion model. Fig 5 illustrates the state transitions of a page *p* for each of the two models. Both diagrams

start with a state where p is neither in the database buffer pool (B) nor in the Memcached (M), denoted by the state labeled as ($p \notin$ B, $p \notin$ M).

Starting from the state of ($p \notin$ B, $p \notin$ M), the next possible event is the receipt of a request for p. In the case of the overflow model, p is brought into B, and the state of p transitions to ($p \in$ B, $p \notin$ M). From this point, the only possible transition is for p to be evicted from B, in that case p is moved to M and the state transitions to ($p \notin$ B, $p \in$ M). From this point, there are 2 possible transitions. The first one is triggered by p being evicted from M, which leads to the initial state of ($p \notin$ B, $p \notin$ M). The second possibility is the arrival of a request for p, which leads to moving p to B from M, and leads to the state of ($p \in$ B, $p \notin$ M).

In comparison, the inclusion model does not have a ($p \in$ B, $p \notin$ M) state, but has a ($p \in$ B, $p \in$ M) state, as shown in the right side of Fig 5.

Which model will deliver better performance depends on workload characteristics. We have implemented both models in our prototyping and conducted preliminary experiments on both models.

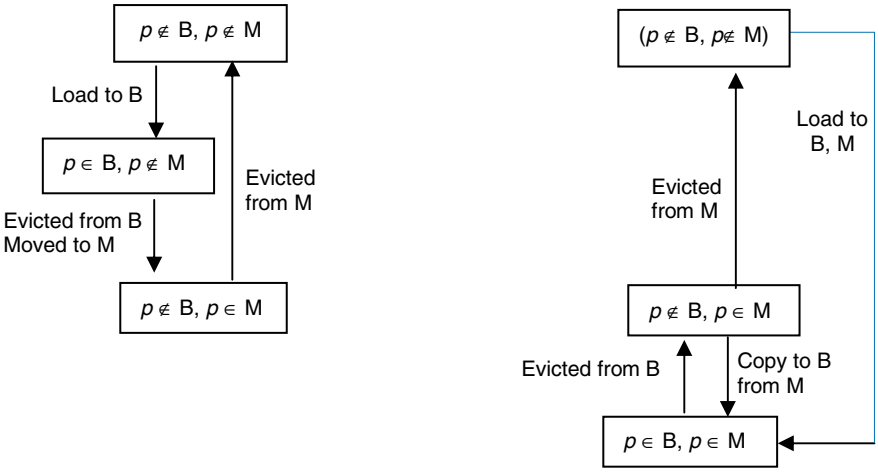


Fig. 5. Transitions of a page state in Overflow model (left) and Inclusion model (right)

We support flexibly configurable page buffering integration policies for fitting various workloads. The algorithms for the basic overflow model and the inclusion model discussed above are switchable and specified in a configuration file. When the PostgreSQL engine starts, the corresponding code sections in the storage module are executed. More detailed analysis and experiments will be conducted to determine the significance of the tradeoffs between the 2 policies.

3.4 Considerations for Stream Query

We have extended the PostgreSQL engine for processing stream data. The details are reported in [4]. Here we only discuss the specific buffer management issues related to stream queries.

In a database system, the result of a query is buffered in the "query result buffer" that is separate from the buffer pool. When the same query is re-issued the buffered result can be delivered directly without re-executing the query. Query result buffers do not have to persisted since the contents can be recomputed upon recovery after a system failure.

Unlike a regular query that uses table-scan to get input data from the underlying database, a stream query uses "function-scan" to capture the input events on-the-fly, e.g. reading from a socket, a log-file, etc. A stream query by nature is a continuous query with infinite input and output; re-running a stream query results in processing newly arrived stream data rather than processing a table statically. In this case the goal of buffering stream query results is for keeping the history of stream processing results.

We combine the buffering of most recent stream query results in a sliding window by pages with the database buffer management in the following way.

- The result of a stream query has a relation schema that is automatically generated by the query engine;
- This schema underlies a virtual table, VT, as a temporary relation mechanism available in the query engine;
- Virtual files and file blocks are derived from the continuous blocks of the resulting data, for the purpose of identifying these blocks (pages). The file ID is associated with a timestamp. Each file block and buffer block (page) has the same size of 8KB. The identifier of the page (i.e. buffer block), composed with the VT's temp-space ID, table ID, file ID and the series number of the block in the file, is serialized to a string key. The query engine's functions for supporting "select into" mechanism are leveraged.

With the above arrangement, the stream query results are buffered in the same way as a regular table, and the buffer can be extended to DCP. On the DCP side, the buffered results of each single stream query are treated as an individual buffer unit with its individual LRU. This is trivial for the DCP systems with nested structure, but requires adding simulated namespace management for Memcached.

When a large number of pages of a stream query are cached in the DCP, which contain the most recent history of stream processing results in a sliding page window, these pages can be retrieved by other regular queries. However, if the stream query does not insert the result chunks explicitly to a regular table, the most recent result pages, although cached in DCP, are not persisted. This is to say that in case persistence is required, inserting the query results to a regular table must be specified explicitly, and in that case the buffered pages are handled in the same way as for regular tables as we described in [4].

These considerations for caching stream query results are still under investigation. More experiments will be conducted in this respect as future work.

4 Preliminary Experimental Results

We have built a prototype MemcacheSQL system based on algorithms and policies explained in the previous sections. In this section we report preliminary experimental results. We illustrate system performance using synthetic data sets simulating different query workloads. We compare between two systems: a conventional

PostgreSQL system with the regular buffer pool management and the MemcascheSQL where the data are additionally buffered on the distributed Memcached servers.

Our testing environment includes 5 Linux servers; all servers run Red Hat Enterprise Linux Server release 5.6. One server has PostgreSQL 8.3.5 installed, 32G RAM, and 8 Quad-Core AMD Opteron Processor 2354 (2.2GHz, 512 KB cache). The other four servers have Memcached installed, 6G RAM, and Quad-Core Intel Xeon 2.8GHZ.

For our experiments, we configure each of the five systems with a buffer cache size of at least $1/4^{th}$ of the database size, while varying the database sizes from 50MB to 10GB. In other words in our current experiments we assume that Memcached is big enough to hold the entire database. The database consists of a single table *T* with 1 million to 200 million tuples, where each tuple is 50 bytes containing 3 attributes: *pid, x,* and *y*. The queries tested are listed below.

Q1: Select sum(T.x) from T;

Q2: Select * from T where T.pid = *v*;

Q3: Update T Set T.x = x+1 T.y=y+1;

Q1 is a sequential scan, Q2 is an indexed retrieval, and Q3 is a sequential scan with update performed on each tuple. All experiments are performed with a warm start; i.e., after the buffer cache on both the PostgreSQL node and the Memcached nodes are filled with data by first running a sequence of sequential scan queries. All results are based on the average of 5 different runs. For experiments that use DCP, the Inclusion Model is first used. We include a performance comparison of the inclusion model and the overflow model at the end of the section.

Experiments of Retrieval Queries. The performance comparison of Q1 (sequential data retrieval i.e., table scan) is given in Fig. 6. The database size ranges from 25M to 200M tuples. The performance without DCP (i.e., "Disk") ranges from an average of 58 seconds to an average of 482 seconds, while that with DCP (i.e.., "DCP") ranges from an average of 38 seconds to an average of 304 seconds. The average performance gain is approximately 1.6x.

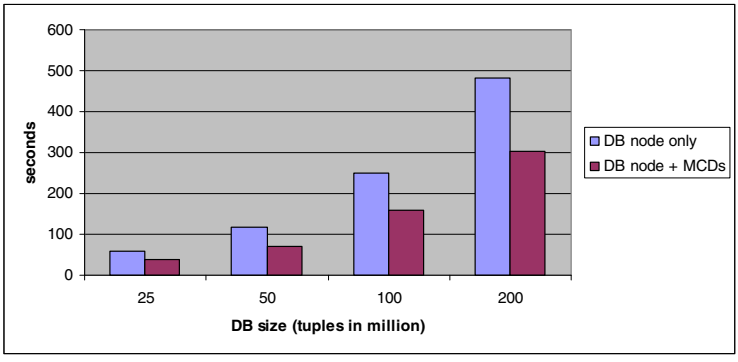


Fig. 6. Query response time comparison: sequential data retrieval from DCP vs. DB Node only

The comparison of Q2 (indexed data retrieval) is shown in Fig. 7. For selectivitiy, we assume that 10 tuples qualify in every 25 million database tuples. The average response time of the query ranges from 265 seconds to 344 seconds when only database node is used, and from 26 seconds to 37 seconds when DCP is used. The performance gain due to the use of DCP is approximately 10x.

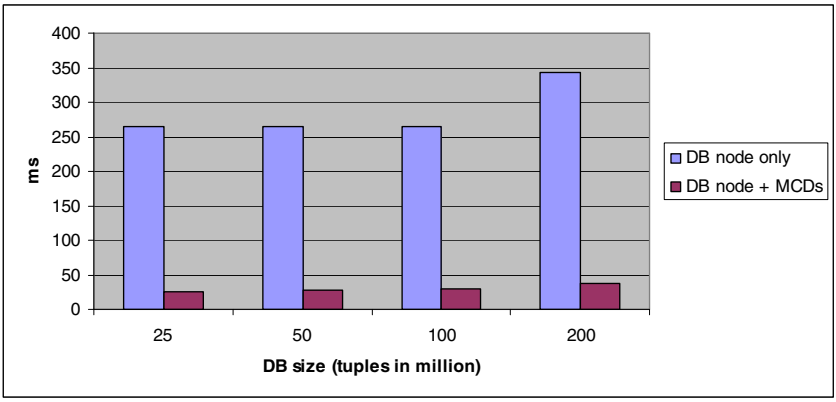


Fig. 7. Query response time comparison: indexed data retrieval from DCP vs. DB node only

Experiments of Update Queries. The performance comparison of Q3 (a sequential retrieval with update) is given in Fig. 8. The average performance gains for varying database sizes range from approximately 3.7x to 4.1x, with an average gain of 3.91x.

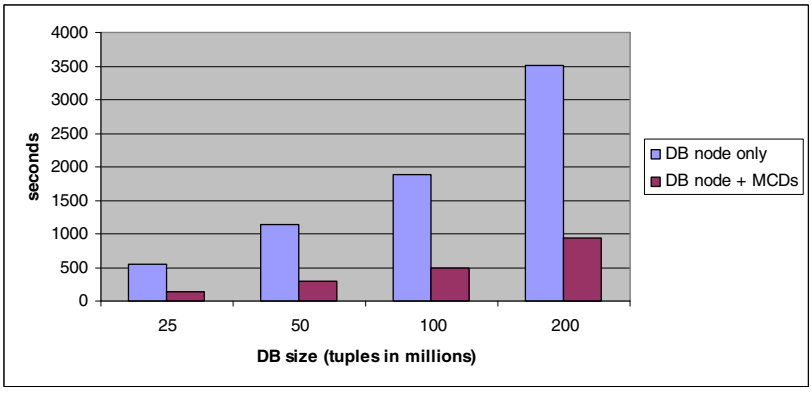


Fig. 8. Query response time comparison: sequential retrieval with update - DCP vs. DB node only

The query performance gain with DCP strongly depends on the query workload characteristics. However these early experimental results indicate that there is good potential with DCP-enlarged buffer pool, alghough additional experiments with real application workloads are needed to further validate the applicability of such an architecture. While there are differences between an in-memory database system and a database system with a sizable buffer pool, where the latter incurs overhead in

buffer management and inter-process communication overhead; however the latter system can scale out to multiple memory nodes, allowing more dynamic allocation of memory resources.

Inclusion Model vs. Overflow Model. Fig 9 compares the data access latency under the inclusion model and the overflow model. In this experiment the table with 25M, 50M, 100M and 200M are first loaded by the sequential scan query Q1 ("First Query Run"), so that data is loaded into the buffer pool and Memcached cache. Q1 is then run again ("Second Query Run"). In First Query Run, the performance of the overflow model is about the same as that of the inclusion model. However, in the second and all subsequent runs, the inclusion model shows a gain up to approximately 18%.

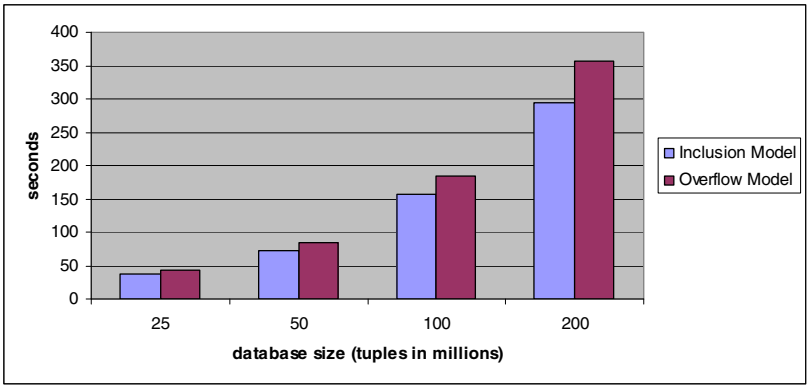


Fig. 9. Comparison of Inclusion Model and Overflow Model

To explain the above experiment results, note that in our experiments, the database size is larger than the buffer pool size, B, and the total cache size of Memcached, M, is larger than the database size. In the initial data loading phase, under the overflow model, most pages loaded to B will eventually "overflow" (evicted and moved) to M, therefore the costs of loading pages to Memcached under the overflow model and the inclusion model are similar. However, after most or all pages already kept in M, under the overflow model, every page evicted from B has to be moved to M; but under the inclusion model, only the dirty pages evicted from B will involve moving the content to M (to refresh the corresponding content in M); for non-dirty pages, only a notification to M is performed. We plan to expand the experimental scenarios to understand the tradeoffs better.

5 Conclusions

In this paper we examine the approach of extending database buffer pool with Distributed Caching Platform (DCP), and demonstrate the scalable in-memory data cache engine, MemcacheSQL, for low latency data access as required by many enterprise applications and data-intensive analytics. This approach allows us, with minimal effort, to combine the provide the scalable MemcacheSQL with full SQL interface, fast data access and ACID properties.

We implemented the proposed system by extending PostgreSQL's buffer management to access the buffered pages cached in distributed Memcached sites, the widely used DCP platform. Our major contributions include data coherence management along the memory hierarchy, and flexibly configurable page buffering policies for best fitting various workloads. Since we ensure all buffered data to be operated under the PostgreSQL buffer management, our MemcacheSQL is robust against failures of any connected Memcached server. Integrating the query engine's buffer management with DCP leverages both the mature DBMS capabilities as well as the memory scale-out capabilities of the DCP technology.

We are continuing our investigations of MemcacheSQL. The next steps include: (a) Experiment with real-world application work loads that motivated the investigation of this architecture, and formally characterize performance tradeoffs among policies and workloads; (b) Incorporate considerations for stream processing; (c) Extend Memcached for supporting several DCP page (key-value) eviction strategies such as clock-Sweep, usage-count and buffer-ring (simulated DCP namespace).

References

[1] Nori, A.: Distributed Caching Platforms. In: VLDB 2010 (2010)

[2] Arasu, A., Babu, S., Widom, J.: The CQL Continuous Query Language: Semantic Foundations and Query Execution. VLDB Journal 15(2) (June 2006)

[3] Abadi, D.J., et al.: The Design of the Borealis Stream Processing Engine. In: CIDR 2005 (2005)

[4] Chen, Q., Hsu, M., Zeller, H.: Experience in Continuous analytics as a Service (CaaaS). In: EDBT 2011 (2011)

[5] Franklin, M.J., et al.: Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In: CIDR 2009 (2009)

[6] Memcached (2010), http://www.memcached.org/

[7] EhCache (2010), http://www.terracotta.org/

[8] Vmware vFabric GemFire (2010), http://www.gemstone.com/

[9] Gigaspaces Extreme Application Platform (2010), http://www.gigaspaces.com/xap

[10] IBM Websphere Extreme Scale Cache (2010), http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.over.doc/cxsoverview.html

[11] AppFabric Cache (2010), http://msdn.microsoft.com/appfabric

[12] Liarou, E., et al.: Exploiting the Power of Relational Databases for Efficient Stream Processing. In: EDBT 2009 (2009)

[13] The Wafflegrid Project, http://www.wafflegrid.com/

[14] http://www.scribd.com/doc/14603868/Distributed-Innodb-Caching-with-memcached

[15] Membase, http://www.couchbase.com/

[16] Baer, J., Wang, W.: On the inclusion properties for multi-level cache hierarchies. In: Proceedings of the 15th Annual International Symposium on Computer Architecture, ISCA 1988 (1988)

A Cost-Aware Strategy for Merging Differential Stores in Column-Oriented In-Memory DBMS

Florian Hübner[1], Joos-Hendrik Böse[2],
Jens Krüger[1], Cafer Tosun[2], Alexander Zeier[1], and Hasso Plattner[1]

[1] Hasso Plattner Institute
[2] SAP AG

Abstract. Fast execution of analytical and transactional queries in column-oriented in-memory DBMS is achieved by combining a read-optimized data store with a write-optimized differential store. To maintain high read performance, both structures must be merged from time to time. In this paper we describe a new merge algorithm that applies full and partial merge operations based on their costs and improvement of read performance. We show by simulation that our algorithm reduces merge costs significantly for workloads found in enterprise applications, while improving read performance at the same time.

1 Introduction

Providing a unified view on real-time data for interactive and ad-hoc data analysis is the goal of column-oriented in-memory DBMS, such as the SanssouciDB outlined in [6]. This DBMS is designed to consolidate the OLTP and OLAP workload of enterprises in one in-memory database, guaranteeing high performance for analytical queries on real-time data as well as for transactional insert operations. While the concept of column-orientation favors mainly analytical query processing [4] and high compression rates [1], changing data in compressed column-structures is expensive. A standard approach to overcome this problem in column-oriented DBMS, e.g. as applied in MonetDB [3] and C-Store [7], is to use an additional write-optimized structure where all changes to the data are accumulated. Since read queries must access both, the read-optimized main store as well as the write-optimized differential store, to derive the consistent state of the DB, overall performance of read queries degrades with increasing size of the latter. To compensate for this performance degradation, changes accumulated in the differential store need to be merged into the read-optimized main store to clear the differential store. Such a merge operation can lead to significant overhead, because it may involve changing the dictionary and re-encoding of all values in a column. In an enterprise environment where columns can easily contain hundreds of millions of records, this may reduce system performance significantly. However, integrating the differential store into the main store from time to time is crucial to maintain viable read performance on frequently updated tables.

M. Castellanos, U. Dayal, and W. Lehner (Eds.): BIRTE 2011, LNBIP 126, pp. 38–52, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In this paper we present a merge strategy that significantly reduces merge overhead for enterprise workloads, while improving read performance at the same time. Specifically, our contributions are the following:

1. We analyse the value distribution of inserts in enterprise systems and present the range of distributions found in typical enterprise applications.
2. We present a merge strategy based on the *partial merge operation*, a merge *trigger strategy* and a *cost aware decision* between full and partial merge operations.
3. We evaluate our approach with a simulation study.

The paper is structured as follows: Section 2 defines our system model and introduces our cost metrics for merge, insert, and select operations. Section 3 presents an analysis of value distributions in columns and their influence on the dictionary encoding in column-oriented data structures. Section 4 gives a detailed description of our algorithms, which is evaluated with a simulation study in Section 5. In Section 6 we conclude and summarize our findings.

2 Architecture, System and Cost Model

The abstract design of a table in SanssouciDB is depicted by Figure 1. Both structures, the read-optimized main store and the write-optimized differential store rely on dictionary encoding [1] for compression. Each column in the main store consists of a column vector C_M and a dictionary D_M. D_M defines a bijective mapping $valueID \leftrightarrow value$, with $valueID$ simply being the position of $value$ in the dictionary vector. Values in D_M are ordered. The number of bits required to encode an entry in C_M is given by $\log_2(|D_M|)$, with $|D_M|$ being the number of distinct values in C_M. A column of the differential store also consist of a column vector C_D and a dictionary D_D. In contrast to the main store, the differential store's dictionary D_D is unsorted but uses a CSB+ tree based index structure to translate value ids to values with logarithmic complexity as shown in Figure 1.

2.1 Merge Cost Model

In order to maintain acceptable read performance over time, the main and differential data structures have to be merged. As described in [6], the *merge algorithm* combines the two dictionaries D_M and D_D to a new dictionary D'_M and then combines C_M and C_D with respect to it. Merging the two dictionaries can be done in $O(\mid D_M \mid + \mid D_D \mid)$ by performing a merge sort while traversing both in order. However, this process can trigger two events influencing the combination of the two columns C_M and C_D. First, the number of bits needed to represent all distinct values in D'_M may be increased by the additional values from D_D. Then every value in C_M has to be updated in order to provide additional bits per entry. We call this event *dictionary overflow* Ξ. It happens whenever the number of distinct values is doubled and becomes more and more infrequent with a growing D_M due to the larger steps and also the mature dictionary with

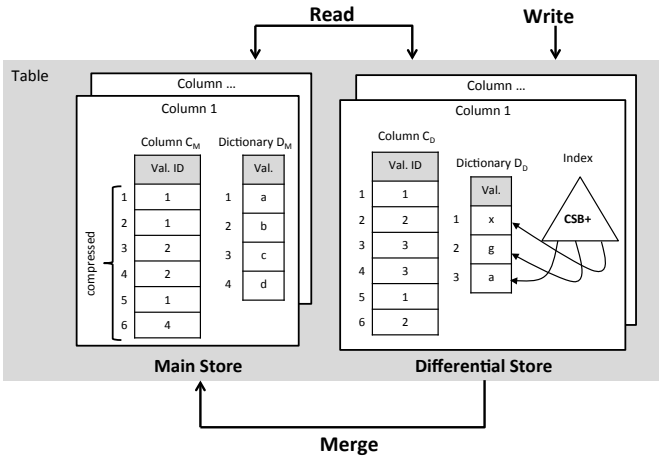


Fig. 1. Data structures of main and differential store

regard to inserted values. Second, in order to preserve the ordering within the dictionary, any value $v \in D_D$ has to be inserted into D'_M before every $u \in D_M$ with $v < u$, hence the value id for each of those u has to be updated — not only in the dictionary but for every entry representing u within the main store column C_M, resulting in a complexity of $O(\mid C_M \mid + \mid C_D \mid)$ (note that the cost for lookups in the dictionaries can be avoided by creating a mapping between the old and the new dictionary while merging D_M and D_D). We call this event *dictionary re-ordering* Γ.

In practice, whenever C_M and C_D are being combined to a new main column, the original copy of the complete main column is held in memory to enable concurrent read operations while updating the encoding of the new C_M. This implies that the amount of memory needed for the database may be doubled when merging and also that every merge operation leads to $\mid C_M \mid + \mid C_D \mid$ write operations, resulting in the invalidation of all caches holding the column's data. Therefore, we count the number of write operations needed to perform a merge as a proxy for its overall cost. It depends on the current status of the main and differential structure and is defined as follows.

$$mergeCost = \mid D_M \cup D_D \mid + \mid C_M \mid + \mid C_D \mid \tag{1}$$

The sizes of C_D and D_D have to be optimized in order to be small enough to achieve acceptable response times for queries despite the unordered dictionary, and as large as possible in order to have as few merge operations as possible. The latter objective is underlined by the observation that the number of merge operations performed before any given point in time has no effect on the cost of future merge operations as all summands are invariant.

2.2 Select and Insert Cost

In order to objectively evaluate the performance of our in-memory database system, we define measures that separately look at insert and select operations and discuss their effect on typical workloads. Insert operations are meant to insert single values into a column. Select operations can be divided into point selects and range selects. As different approaches to organizing columns and dictionaries within this paper do not affect the DB's operators after the selected values have been found in a column, we do not further investigate their behavior.

Insert Cost. Looking at the complexity of insert operations and taking into consideration what we already know about effects on a dictionary encoded column, it becomes clear that the cost for inserting a single value is strongly dependent on all previous insert operations. When inserting a single value, it is first added to the differential structures. Looking up the value in D_D is of logarithmic complexity $O(log_2(\mid D_D \mid))$ due to the CSB+ index, and adding the found value id to C_D requires a single write operation. If a merge event is triggered, it results in another $\mid D_M \cup D_D \mid + \mid C_M \mid + \mid C_D \mid$ write operations with regard to the cost defined in (1). To evaluate the overall insert performance, we count the number of write operations needed to perform n single insert operations of the values $\{v_1, \ldots, v_n\}$ and evaluate the amortized figures over time. With $writes(v_i)$ being the number of write operations needed to insert v_i including the cost for any merge event triggered before the next insert, we define the amortized insert cost per value $insertCost(n)$ after n inserts as follows.

$$insertCost(n) = \frac{\sum_{i=0}^{n} writes(v_i)}{n} \tag{2}$$

Select Cost. A point select operation is a query that returns any entry of a column (or its position) whose attribute value x equals a given value v. It has to be performed on the main and the differential data structures. Looking up the value id of v in D_M and D_D requires $\log_2(\mid D_M \mid)$ and $\log_2(\mid D_D \mid)$ read operations each. If the value id is found in either dictionary, the respective columns have to be scanned requiring $\mid C_M \mid$ and $\mid C_D \mid$ compare instructions (we assume, that no inverted index exists, as we opperate on any given column of a transactional DBMS). It can be seen that the number of operations needed to perform a point select operation is the same within the main and the differential data structures and therefore will be neglected for the purpose of this paper.

A range select operation is a query that returns any entry of a column (or its position) whose attribute value x lies within a given interval defined by its minimum value v_{min} and its maximum value v_{max}. To perform the query on the main data structures, first the interval $[v_{min}, v_{max}]$ has to be looked up in D_M. The smallest value $v_i > v_{min}$ and the largest value $v_j < v_{max}$ and their value ids id_{min} and id_{max} can be found with $2 \cdot \log_2(\mid D_M \mid)$ compare operations. After that, the column C_M is scanned and for each value id i it has to be checked, whether $id_{min} \leq i \leq id_{max}$ holds, resulting in $2 \cdot \mid C_M \mid$ compare instructions.

Performing the same operation on the differential data structure is more costly. As the dictionary D_D is unsorted, the interval $[v_{min}, v_{max}]$ can no longer be represented by two values, but all values $u \in U = [v_{min}, v_{max}] \cap D_D$ and their value ids have to be calculated in $O(\mid D_D \mid)$. Then C_D has to be scanned, but instead of 2 compare instructions per value v, a lookup to the previously calculated value set has to be made. Using the value ids as offsets to a respective data structure the computational complexity stays unchanged, however, an extra layer of indirection is added. The overall cost of a range query is defined by the number of compare operations on the one hand and the number of necessary memory accesses (to the different levels of the memory hierarchy) on the other hand and has to take specifics of the underlying hardware into consideration, represented by scaling factors s_{cmp} for compare instructions and s_{read} for random memory accesses. For the purpose of this paper we neglect the logarithmic look-up cost and calculate the cost of a range select query as follows:

$$selectCost = 2 \cdot \mid C_M \mid \cdot s_{cmp} + 2 \cdot \mid D_D \mid \cdot s_{cmp} + \mid C_D \mid \cdot (s_{cmp} + s_{read}) \quad (3)$$

Previous work shows that OLTP as well as OLAP enterprise workloads comprise of mainly read operations. E.g. [5] mentions that even in OLTP workloads logged on enterprise systems about 84% of all operations are read queries, in OLAP workloads this increases to even 93%. In the context of this article we are especially interested in the fraction of range selects among read queries, since the performance of range queries degrades with a growing $|C_D|$. From [5] we obtained that in a typical enterprise OLTP workload 30% of all queries select a range of values, while in a OLAP environment 44% of all queries are range selects. Hence, it can be stated that the performance of range selects is crucial to the overall database performance in enterprise workloads.

3 Data Characteristics of Enterprise Systems

The complexity of merge operations depends on the distribution of values represented by D_M and D_D at merge time. Our algorithm improves merge complexity by leveraging characteristics of these distributions. To evaluate the benefit of our algorithm in a typical ERP environment, we derived the characteristic value distributions of columns in enterprise applications found in SAP's Business Suite [2].

We analyzed the distributions of values within columns in SAP's financial (FI) and in the sales and distributions (SD) module of the Business Suite. We examined tables of both applications in two customer installations, while only the master and line items tables were considered, since these are the most frequently accessed tables. In total we analyzed 1864 columns. For every column we computed the frequency of each value and ranked them accordingly in the resulting histogram. In a first step we excluded all histograms, which had more than one value and every value occurs exactly once. For these key or id columns

dictionary encoding would not be applied. Also, columns with only one distinct value were not considered any further, as they allow for avoiding most of the costs for inserts and queries by engineering.

The remaining histograms (in total 770) have more complex value distributions and thus induce more complex computations for calculating query results. In order to describe their characteristics more generally, they were partitioned into two sets: (i) columns with a distribution best approximated by a zipf pdf, and (ii) columns where a uniform distribution provides a smaller rooted mean square error (RMSE) than a zipf pdf. Both pdf functions rely on a given number of distinct values as a parameter. The zipf distribution was chosen because we recognized that a lot of columns of FI and SD follow a power-law distribution, i.e., a small set of values occurs frequently, while the majority of values (long tail) are rare.

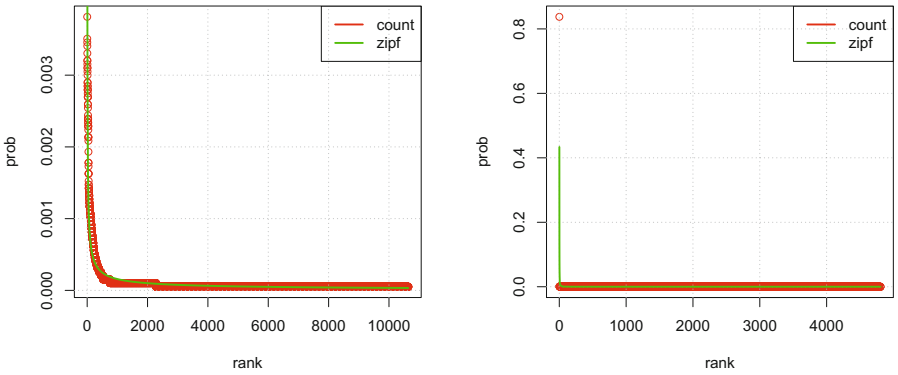
Table 1. Value distributions and their parameter five-number summaries

pdf	***fraction***	***parameter range***	***RMSE***
zipf	21.03%	$\alpha_{min} = 0.001$ $\alpha_{Q1} = 1.030$ $\alpha_{Q2} = 1.581$ $\alpha_{Q3} = 2.586$ $\alpha_{max} = 4.884$	$e_{min} = 0.000$ $e_{Q1} = 2.599e^{-16}$ $e_{Q2} = 3.234e^{-15}$ $e_{Q3} = 3.095e^{-13}$ $e_{max} = 7.630e^{-13}$
uniform	20.02%	$k_{min} = 2$ $k_{Q1} = 2$ $k_{Q2} = 5$ $k_{Q3} = 86$ $k_{max} = 216$	$e_{min} = 5.749e^{-17}$ $e_{Q1} = 3.116e^{-16}$ $e_{Q2} = 2.572e^{2}$ $e_{Q3} = 1.465e^{5}$ $e_{max} = 2.398e^{5}$

Table1 summarizes some results of our analysis. It shows that 51% of the relevant columns (i.e., 21.03% of all columns analyzed) are best approximated by a zipf distribution, while 49% are best approximated by a uniform distribution. For those zipf and uniform distributions Table1 gives the five-number summary of the parameter range. The five-number summary for the zipf parameter α shows a range $[\alpha_{min} = 0.001, \alpha_{max} = 4.884]$, with a median at $\alpha_{Q2} = 1.581$. We also show the RMSE of the fitted distributions.

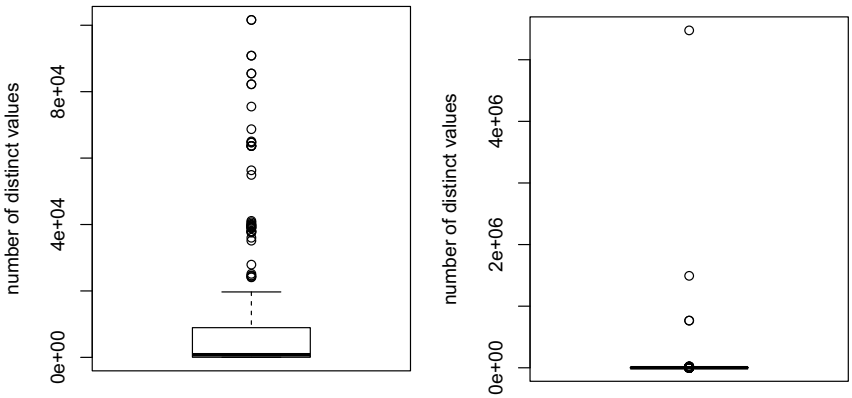
Figure2b) shows the ranked histogram for one analyzed column with a zipf coefficient close to α_{Q2}. Figure2a) gives another example with zipf parameter α=0.651. Both examples show the typical power-law distribution of values: there are some values that occur frequently, while the majority is rare. Our algorithm will especially leverage the power-law characteristics of such distributions to reduce merge costs.

Besides the zipf parameter α, the total number of distinct values in a column influences the distribution and is the single parameter of uniform distributions. Figure3a) depicts the total number of distinct values for all columns that follow a zipf distribution, while Figure3b), depicts the same for columns with uniform distributed values using a boxplots.



(a) $\alpha = 0.651$ and $n = 10642$ (b) $\alpha = 1.588$ and $n = 4806$

Fig. 2. Histograms and zipf fits



(a) zipf (b) uniform

Fig. 3. Number of distinct values

4 Cost-Aware Merge Strategy

In order to reduce the costs for merging values from the differential data structures described in Section 2 into the main store data structures, we propose an algorithm that is able to merge only a subset of D_D into D_M so that the gain in performance is high but the cost for merging is significantly lower in total. The main idea is to decide the largest subset of values in D_D and C_D that does not trigger expensive re-ordering events Γ when being inserted into D_M (see

Section 2.1). As a consequence, it becomes necessary to decide between performing a *partial merge* and a *full merge* whenever a merge event is triggered. In the following, we describe the partial merge algorithm in detail as well as strategies when to trigger merge events and to decide between partial and full merge.

4.1 Partial Merge Algorithm

The partial merge algorithm as described in Algorithm 1 decides the largest subset of values in D_D and C_D that can be merged into the main store data structures D_M and C_M without making the expensive re-ordering of the dictionary and the implied re-encoding of the main column C_M necessary (see Section 2.1). This is achieved by only merging all values v that either are already contained in D_M or whose rank in the ordering of D_M is higher than the ranks of all other values in the dictionary. If v is already contained in D_M, the corresponding value id $valueID(v)$ will be used to encode v in C_M. If the value v is not contained in D_M but of greater rank than all values in D_M, then $valueID(v)$ can be added at the end of D_M without having to update any information in D_M. In both cases, the encoded value can be added at the end of the main store column C_M without affecting all previously inserted values, i.e., no re-encoding is necessary. Finding the set of all value ids *mergeVals* that can be merged without triggering a re-ordering event can be done in $O(\mid D_M \mid + \mid D_D \mid)$ by traversing both dictionaries' values in order. At the same time, we calculate a mapping from D_D to D_M and mark all values contained in *mergeVals*. For each value in C_D we then check, whether it is contained in *mergeVals* and either add it to C_M or to the updated C_D, all of which can be done in $O(\mid C_D \mid)$. The overall complexity of the partial merge algorithm with respect to the columns and dictionaries used for the main and differential store results to $O(\mid D_M \mid + \mid D_D \mid + \mid C_D \mid)$. This is significantly faster compared to the complexity of the full merge in $O(\mid D_M \mid + \mid D_D \mid + \mid C_M \mid + \mid C_D \mid)$ as described in Section 2.1. It can be seen that the complexity of the partial merge is no longer related to the size of the main column. Therefore, the frequency for triggering a partial merge in order to optimize read performance can be significantly higher without compromising on the overall performance.

4.2 Merge Trigger Strategies

The responsibility of a trigger strategy is to monitor the state of the differential store and to initiate a merge to maintain viable *selectCost*. In general, the cost overhead for merging has to be compensated by the cost saved on select queries, therefore the distribution of queries in the workload influences the choice of trigger parameters. The strategy proposed here aims at keeping read-costs at a low level and is called *fraction of read optimum (fro)*. In Section 5 we will evaluate this strategy against the naive approach, which is called *fraction of main* (*frm*).

```
   Data: C_M, D_M, C_D, D_D
   Result: Updated columns and dictionaries
 1 // find values to merge by comparing dictionaries D_M and D_D
 2 // and calculate mapping between the dictionaries
 3 mergeVals = []
 4 mapping = []
 5 i_m = 0
 6 i_d = 0
 7 while i_m < size(D_M) and i_d < size(D_D) do
 8     if D_M[i_m] == D_D[i_d] then
 9         mergeVals.add(D_D[i_d])
10         mapping[i_d] = i_m
11         i_m++
12         i_d++
13     else if D_M[i_m] > D_D[i_d] then
14         i_d++
15     else if D_M[i_m] < D_D[i_d] then
16         i_m++
17     end
18 end
19 if i_m = size(D_M) and i_d < size(D_D) then
20     for i_d = i_d to size(D_D) − 1 do
21         mergeVals.add(D_D[i_d])
22         mapping[i_d] = i_m++
23         D_M.add(D_D[i_d])
24         D_D.remove(v)
25     end
26 end
27 // find values in differential column and attach to main column
28 for each v in C_D do
29     if v in mergeVals then
30         C_M.add(v)
31         C_D.remove(v)
32     end
33 end
```

Algorithm 1: Partial Merge Algorithm

Fraction of Read Optimum (fro). The basic idea of this strategy is to maintain a read-cost level that has a defined distance to the theoretical optimal read-costs. We define the optimal read-costs after n inserts, denoted by $selectCost^{opt}(n)$, as the worst case costs of a range query with $|C_D| = 0$, i.e., $selectCost^{opt}(n)$ describes the maximum costs of a range query when the differential store is fully merged into the main store. The fro strategy allows $selectCost(n)$ to grow until it exceeds a defined fraction f_{opt} of $\text{selectCost}^{opt}(n)$. Thus, a merge

is triggered every time (4) is true. As $selectCost(n)$ and $selectCost^{opt}(n)$ are defined by the sizes of the data structures, both values can easily be calculated at runtime.

$$selectCost(n) > f_{opt} \cdot selectCost^{opt}(n) \tag{4}$$

Fraction of Main (frm). When applying the *fraction of main (frm)* strategy, a merge is triggered, whenever $|C_D|$ reaches a threshold size s_t relative to $|C_M|$.

$$s_t \cdot \mid C_M \mid < \mid C_D \mid \tag{5}$$

4.3 Full Merge vs. Partial Merge

We leverage the performance functions defined in Section 2.2 to choose between a full and a partial merge whenever a merge event is triggered. If the expected performance after a partial merge $selectCost_P$ is comparable to that after a full merge $selectCost_F$, a partial merge will be executed. We define the factor f_{to} to represent the tolerated overhead with regard to the merge costs saved, thus a partial merge is performed whenever (**??**) evaluates true, a full merge otherwise. If a merge causes a dictionary overflow Ξ, we always choose a full merge operation.

$$selectCost_P \leq f_{to} \cdot selectCost_F \tag{6}$$

$$2 \cdot (\mid C'_M \mid + \mid D'_D \mid) \cdot s_{cmp} + \mid C'_D \mid \cdot (s_{cmp} + s_{read}) \leq f_{to} \cdot 2 \cdot (\mid C_M \mid + \mid C_D \mid) \cdot s_{cmp} \tag{7}$$

The optimal value of f_{to} is dependent on the workload. With increasing ratio between range queries and insert operations, the overhead induced by sub-optimal select performance gains influence on the overall database performance, thus the frequency of full merges should be increased by decreasing f_{to}.

Calculating $selectCost_P$ at runtime requires maintaining a list of all values that are merged during a partial merge. Whenever a value $v < max(D_M)$ is added to D_D it has to be looked up in D_M. Additionally, the number of values for each value id in D_D has to be remembered. The total cost for this is in $O(log_2(\mid D_M \mid))$ amortized over the number of inserts between two merge events.

5 Evaluation

We simulate the behavior of the database for inserts sampled from the distributions described in Section 3. To evaluate the performance of our cost-aware merge strategy introduced in Section 4, we compare it to the behavior of the standard algorithm described in Section 2.1 and analyze the insert cost over time as well as the worst case costs for range select queries after each insert (see Section 2.2).

5.1 Simulation Environment and Parameters

The simulation has been implemented in R. We leverage the median parameters from the value distributions shown in Section 3 in order to generate a set of

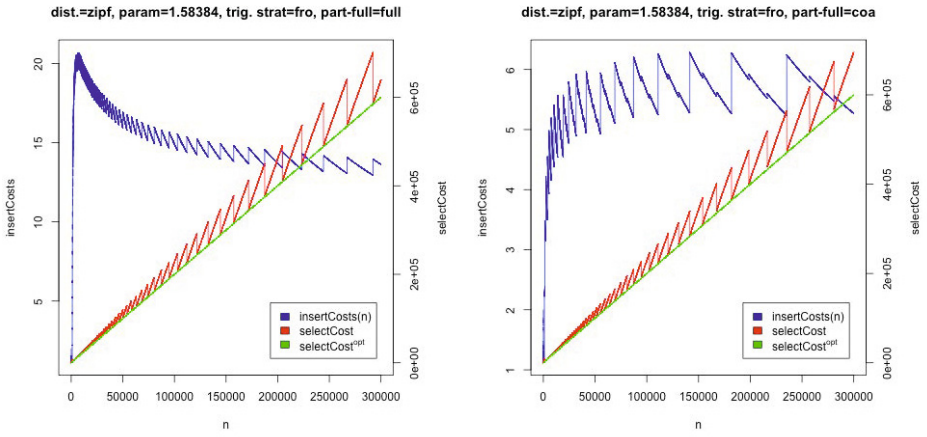
$n = 300k$ values. 6403 distinct values are generated for the zipf distribution as well as 216 distinct values for the uniform distribution. We then simulate the insertion of the values v_1 to v_n and keep track of lists representing the status of the columns C_M and C_D as well as the dictionaries D_M and D_D over time. For the fraction of optimum (fro) trigger strategy we set the threshold factor $f_{opt} = 2.0$, guaranteeing that the select cost will never be more than two times as high as in a fully read optimized architecture without differential store. The fraction of main (frm) trigger strategy applies a threshold size of 10%, i.e., a merge is triggered whenever the differential column's size $\mid C_D \mid$ is 10% of the main column's size $\mid C_M \mid$. In order to decide between a full and a partial merge in case of our cost-aware algorithm, we set $f_{to} = 1.2$, i.e., a partial merge will be performed as long as the read costs after the merge are not more than 20% higher than they would be after a full merge. For the hardware specific scaling factors we chose $s_{cmp} = 1$ and $s_{read} = 5$.

5.2 Analysis of Simulation Results

The simulation study shows that for common distributions observed in enterprise environments, write costs are reduced significantly while guaranteeing upper bounds on read performance by our cost-aware merging strategy. To show the potential reduction in costs for zipf distributed columns, we plot the costs for two representative zipf distributions in Figures 4 and 5.

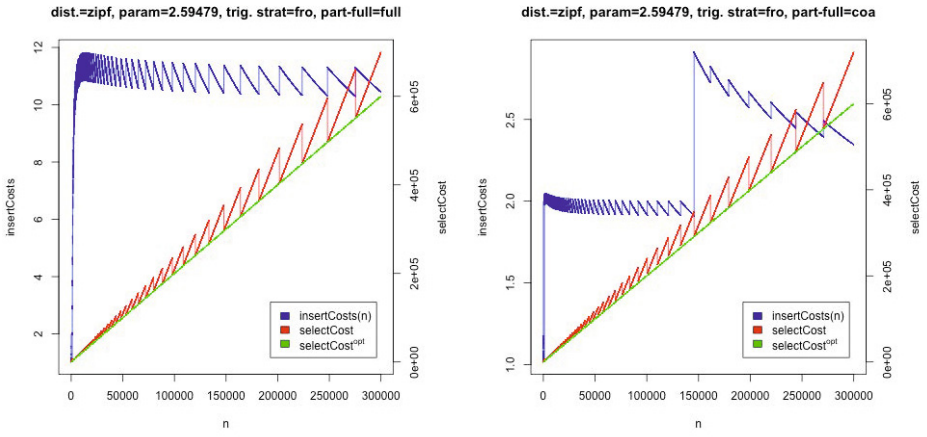
Figure 4 shows the improvement of our algorithm for a workload sampled from a zipf distribution parameterized with the median $\alpha_{Q2} = 1.58384$ of zipf parameters found in our customer data analysis in Section 3. The standard approach with fro trigger and exclusive full merges leads to write costs shown by the blue curve in Figure 4a) and worst case costs of range selects as depicted by the red curve. The green line marks the optimal read costs as computed by $selectCost^{opt}(n)$. *insertCosts* increase fast and stabilize around 14 write operations per insert.The decreasing trend in *insertCost* is due to the effect that for larger merge intervals the chance increases that values inserted are already contained in D_D and can be inserted into C_D with a single write operation instead of two writes. *selectCost* increases with the number of inserts, since more values remain in C_D before a merge is triggered. The red line in Figure 4b) depicts the effectiveness of the partial merge strategy, every partial merge moves the *selectCost* close to the optimum, while causing little extra write costs. It can also be seen where the cost-aware merging strategy decides to perform a full merge instead of a partial merge due to the expected improvement in *selectCost*, as the gap to the optimal *selectCost* is closed and the amortized *insertCosts* are increased by 1.

Figures 5a) and b) plot *insertCost* and *selectCost* for a workload sampled from a zipf distribution with a steeper shape parameter $\alpha_{Q3} = 2.59479$ as derived in Section 3, i.e., fewer frequent values and more rare values in the long tail. Since here the probability that values are already contained in D_D and D_M is higher, *insertCost* does not exceed 3 in our approach as shown in Figure 5b), while in



(a) frm trigger and full merge (b) fro trigger and cost-aware merge

Fig. 4. Development of *insertCost* and *selectCost* for a zipf distributed workload

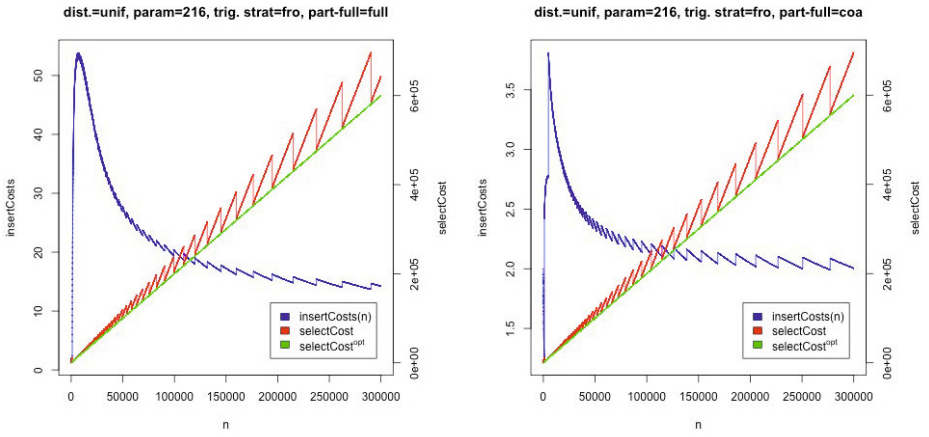


(a) frm trigger and full merge (b) fro trigger and cost-aware merge

Fig. 5. Development of *insertCost* and *selectCost* for a zipf distributed workload

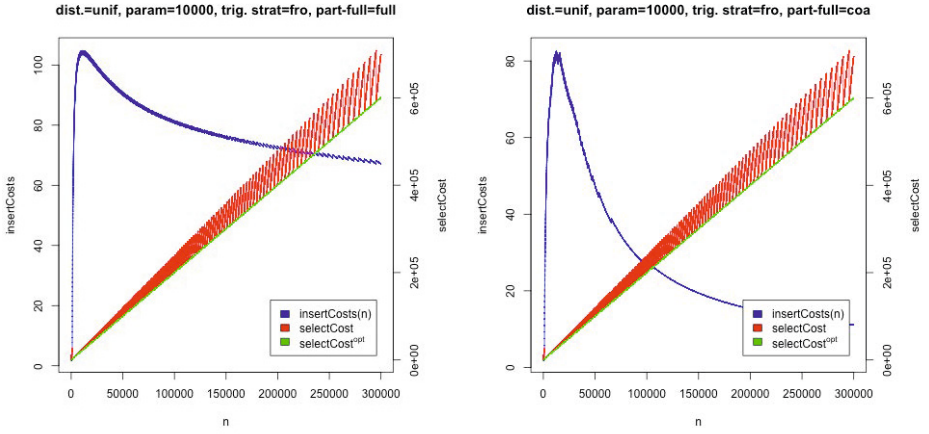
the default approach *insertCost* stays above 10 and is not significantly affected by the distribution as it can be seen in Figure 5a).

To evaluate the performance of our approach for workloads sampled from uniform distributions, we plot *insertCost* and *selectCost* for a uniform distribution with 216 values, which is the $Q3$ parameter derived in Section 3. Due to the small number of distinct values, D_M is saturated quickly and a partial merge operation can insert all values into C_M, resulting in optimal *selectCost* after every merge operation as shown in Figure 6b). A drift away from optimal *selectCost* after



(a) frm trigger and full merge (b) fro trigger and cost-aware merge

Fig. 6. Development of *insertCost* and *selectCost* for a uniform distributed workload with 216 (k_{max}) distinct values



(a) frm trigger and full merge (b) fro trigger and cost-aware merge

Fig. 7. Development of *insertCost* and *selectCost* for a uniform distributed workload with 10000 distinct values

partial merges cannot be observed here. Write costs also benefit from the fact that dictionaries are stable, resulting in nearly optimal insert costs for large n with a peak of 3.8 for small n as shown in Figure 6b), while the naive approach causes *insertCost* of stabilizing at about 12. To demonstrate the effect of increased value domain sizes, we plot *insertCost* and *selectCost* for a uniform distribution with 10000 distinct values in Figure 7. Here, our strategy decides on full merges

frequently, resulting in higher *insertCost* for small n than in the default approach, while for large n amortized write costs are reduced to about 10 for our approach vs. 70 for the default strategy, as depicted in Figure 7 a) and b).

6 Conclusion

In this paper we analyze the behavior of read-optimized column-oriented in-memory databases that leverage a differential store for increasing write performance and therefore face the situation of having to merge the dictionary-compressed main and differential data structures to maintain viable read performance. After examining the complexity of the algorithms used by current databases, we analyze customer data from real-world enterprise systems and present the parameter ranges of zipf and uniform distributions of columns in the relevant tables. We build on the insight how those value distributions influence the dictionaries used for compressing the data structures and introduce the partial merge algorithm, which merges only those values of a column, which do not trigger costly re-ordering events. We integrate the partial merge algorithm into the cost-aware merging strategy, which adaptively triggers merge events and decides between performing a partial merge and the classic full merge algorithm with respect to defined guarantees on amortized insert costs and induced select costs. We show the potential of our cost-aware merging strategy by simulation of database operations using the parameters extracted from the analyzed customer data and comparing it to the classic approach used at present. It can be seen that our approach significantly reduces the amortized insert cost while guaranteeing range select performance at the same time.

As part of our future work, we plan to optimize the cost-aware merging strategy with regard to the workload of the database and more accurate latency of its operators, in order to be able to find a global cost minimum or adapt to service levels for inserts or selects. This will include the analysis of additional customer data in order to add further detail to our current results. Furthermore, we want to investigate the consequences of bringing the concept to tables that may present with different distributions of distinct values for each column. While this work was solely based on simulation, we plan to implement and evaluate our approach in a research prototype.

References

1. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 671–682. ACM (2006)
2. SAP AG. Sap business suite, http://www.sap.com/solutions/business-suite/index.epx
3. Boncz, P.A., Zukowski, M., Nes, N.: Monetdb/x100: Hyper-pipelining query execution. In: Proc. CIDR, vol. 5. Citeseer (2005)

4. Harizopoulos, S., Liang, V., Abadi, D.J., Madden, S.: Performance tradeoffs in read-optimized databases. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 487–498. VLDB Endowment (2006)
5. Krueger, J., Kim, C., Grund, M., Satish, N., Schwalb, D., Chhugani, J., Zeier, A., Dubey, P., Plattner, H.: Fast updates on read-optimized databases using multi-core cpus. In: Proceedings of the VLDB Endowment (to appear, 2012)
6. Plattner, H., Zeier, A.: In-Memory Data Management - An inflection point, vol. 1. Springer (2011)
7. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al.: C-store: a column-oriented dbms. In: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 553–564. VLDB Endowment (2005)

Microsoft SQL Server Parallel Data Warehouse: Architecture Overview

José A. Blakeley, Paul A. Dyke, César Galindo-Legaria, Nicole James, Christian Kleinerman, Matt Peebles, Richard Tkachuk, and Vaughn Washington

Microsoft Corporation, 85 Enterprise Dr. 2nd Floor, Aliso Viejo, CA 92656, USA
joseb@microsoft.com

Abstract. This paper presents an architecture overview of the Microsoft SQL Server Parallel Data Warehouse (PDW) DBMS system. PDW is a massively parallel processing (MPP), share nothing, scale-out version of SQL Server focused on data warehousing workloads. The product is packaged as a database appliance built on industry standard hardware. We present an overview of the main software and hardware components of the system and an example illustrating its query processing approach.

Keywords: Distributed database systems, data warehousing, massively parallel processing, database appliance, distributed query processing.

1 Introduction

Microsoft SQL Server, like most commercial relational database management systems (RDBMSs), was architected in the mid 1990's using techniques developed 40 years ago by database research prototypes such as System R [1] and Ingres [2]. These systems were built to address the needs of an enterprise database market that combined online transaction processing (OLTP) and data warehousing (DW) workloads. To become a serious contender in that market Microsoft built the SQL Server product to address the needs of both workloads.

Today, the OLTP and DW market segments have each grown substantially making it feasible to build database systems focused on the specific needs of each of these workloads [3]. In fact, in the early 2000's many database company start-ups emerged offering scale-out, MPP database engines that addressed the specific needs of large-scale DW workloads (e.g., ParAccel, Netezza, Greenplum, Exasol, DatAllegro, and Vertica). These specialized engines achieve much higher performance on DW workloads than the general-purpose MPP database engines and created a major disruption in the DW market segment. These gains are obtained by leveraging hardware trends (e.g., larger inexpensive RAM, multi-core processors, higher network and memory bandwidths); by using new storage formats (column stores and data compression); and query processing techniques tuned for DW workloads. Another interesting trend was the emergence of the database appliance. These are database systems that are packaged as a complete hardware and software solution.

M. Castellanos, U. Dayal, and W. Lehner (Eds.): BIRTE 2011, LNBIP 126, pp. 53–64, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In mid-2008 Microsoft declared its entrance into the large-scale DW database market by acquiring DatAllegro, a database startup that offered a large-scale MPP database appliance built on Ingres and Linux. Combining DatAllegro and Microsoft technology, the PDW team engineered a large-scale MPP database appliance built on SQL Server and Windows. The first release of the Microsoft SQL Server PDW appliance shipped in October 2010 with HP as hardware partner. Today, there are SQL Server PDW appliance offerings on Dell and HP hardware.

Assembling a high-performance, large-scale, distributed, MPP DBMS system with a properly balanced choice of hardware (processors, RAM, storage, network) is a complex task that involves hundreds of hardware and software design choices. Packaging a DBMS system of this complexity as an appliance offers the benefit of a plug-and-play, low-TCO solution to customers. An appliance customer only has to power-on the system, run an automatic validation to ensure the system is properly configured, create a database, load the data, and start running queries on the system.

This paper presents an overview of the SQL Server PDW appliance. Section 2 introduces the overall appliance including the main hardware and software components. Section 3 describes the layered approach used in building SQL Server PDW and its data distribution policies. There are many interesting software components in an MPP database (e.g., transaction management, query processing, bulk data loading, backup/restore, appliance health, and appliance expansion). In Section 4 we provide a brief description of the SQL Server PDW distributed query processor. Section 5 provides brief description of other key appliance components. Finally, Section 6 presents the summary and conclusions.

2 Parallel Data Warehouse

SQL Server PDW is a scale-out appliance using a shared-nothing, MPP approach to deliver high-performance and scalable data management for DW workloads. The appliance consists of a distributed cluster of computer servers (nodes) performing distinct roles in delivering the solution. The capacity of the system scales from tens of terabytes to over 500 terabytes of raw tabular data. The appliance encompasses software, hardware, high-speed networking, and storage all pre-configured and balanced for the processing of statements typical of DW workloads. A SQL Server PDW appliance is designed to be used without the need of any special tuning, guaranteeing low total cost of ownership for customers. SQL Server PDW achieves high performance and scalability by partitioning user data across and within compute nodes and designing every component of the system for full utilization during the execution of SQL statements.

2.1 Hardware Components

The appliance hardware is based on reference architectures designed and specified by the SQL Server PDW engineering team. These are assembled by independent hardware vendors (IHVs) working in partnership with Microsoft. Each appliance

consists of one or more racks of servers, storage, and networking components. Internally, the appliance consists of a collection of computer servers (nodes), each of which performs specific functions to be described below. The hardware capabilities of each node are sized according to the functions running on the node. The following diagram highlights key aspects of the hardware architecture:

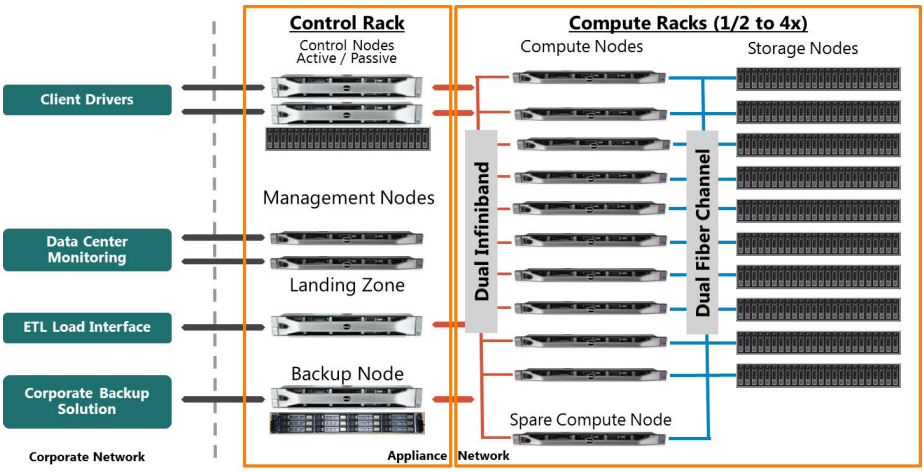


Fig. 1. SQL Server PDW physical architecture

The appliance is a multi-rack system that contains one *control rack* for performing management operations and exposing user entry points as well as one or more *data racks* which store user data and act as the work horse for parallel SQL statement execution.

Control Rack. The control rack houses the Management, Control, Landing Zone, and Backup node functions. The *Management node* is used for administering the appliance as a single system. The Management node (a) hosts an instance of Microsoft Active Directory which defines an internal appliance-wide domain, and (b) acts as the Windows High-Performance Computing (HPC) head node for executing distributed servicing jobs across nodes (e.g., patching and upgrade). The *Control node* runs the PDW Engine process which is responsible for providing a single database system view to clients connected to the appliance. Additionally the Control node hosts Microsoft Internet Information Server (IIS) which exposes a web-based administrator console for monitoring the health of hardware and software components within the appliance as well as providing a dashboard that monitors statement execution in real-time. The *Landing Zone* node is used as a temporary staging store of user data during bulk loads into the appliance as well as to host third-party data analysis software. Applications running on the landing zone benefit from high speed access to user data through an Infiniband network. The *Backup node* stores user data backups and can be optionally housed outside the appliance and managed by an enterprise archival system. All nodes in the control rack are accessible externally from the appliance.

Management and Control nodes have redundancy while Landing Zone and Backup nodes currently do not, but could in the future.

Data Rack. The data rack houses *Compute nodes* and *storage* for user data. The first-generation of PDW appliances support up to 4 data racks each containing ten active and one passive compute nodes in a cluster. Performance and capacity are scaled by adding data racks. Compute nodes are associated one-to-one with a fiber channel SAN storage array. For high-availability, each data rack contains a passive Compute node which can access any of the independent SAN arrays for the purpose of replacing any failed active Compute node.

2.2 Software Components

Consistent with the concept of an appliance SQL Server PDW presents a single database system view over the distributed cluster of nodes for SQL statements and administrative tasks (e.g., backup, restore, bulk load, remote table copy). At the same time, it allows visibility into the physical layout of data and hardware when necessary for administrative operations or hardware maintenance. Figure 2 highlights key software architecture components.

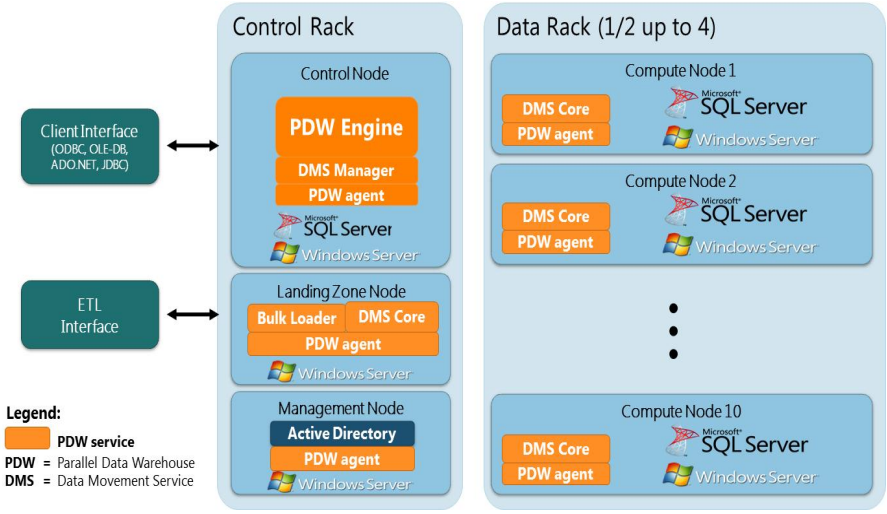


Fig. 2. SQL Server PDW software architecture

Client Tools. PDW provides application connectivity for database development, business intelligence (BI) reporting tools, and bulk data loading through standard database access APIs such as ODBC, JDBC, OLE-DB, and ADO.NET. SQL Server PDW provides out-of-box support for Microsoft BI tools including: Excel Power Pivot; SQL Server Analysis, Reporting, and Integration Services. In addition, it provides connectivity to Informatica, Microstrategy, Business Objects, and SAS Access. The ETL interface also supports connectivity to Hadoop clusters via a SQOOP connector.

PDW Engine. The SQL Server PDW distributed DBMS is architected as a layered system with an explicit separation of concerns between local tasks that execute on a compute or control node and distributed tasks that execute across the entire computer cluster. The PDW Engine service, which runs exclusively on the control node, is a process responsible for providing a single database system view over the distributed collection of SQL Server instances running on compute and control nodes. The PDW Engine uses the Tabular Data Stream (TDS) protocol which provides connectivity to data access APIs; manages authentication and authorization; manages the global (distributed) view of metadata; coordinates T-SQL statement compilation, distributed query optimization, and distributed execution of statements; coordinates T-SQL batch and stored procedure execution; manages global resources (e.g., locks, threads, queues); and coordinates concurrent execution of statements and distributed transactions.

Data Movement Service (DMS). DMS is a collection of collaborating processes responsible for efficient movement of large-volumes of data among nodes within the appliance for distributed query execution and bulk data loads. DMS is essentially a distributed statement execution engine that provides a compact set of primitives required to execute high-performance data movement operations needed during distributed DDL and DML statement execution and bulk loads. The set of primitives supported by DMS are:

- *Shuffle_Move.* This is a many-to-many operation that takes a source distributed table or query result set and redistributes it across the N compute nodes of the appliance based on a hash-partition function computed over a column of the source table or result set. (N:N). The result is a persisted, distributed temporary table.
- *Broadcast_Move*. This is a many-to-many operation that transforms a distributed table into a replicated table. It does this by taking all distributions of a table or result set and broadcasting them to all nodes of the appliance. The result table on each node is the union of all source table distributions.
- *Partition_Move*. This is a many-to-one data movement operation from a set of compute nodes to the control node. It performs the union of distributed partitions across all compute nodes into a single table on the control node.
- *Shuffle_Load*. This is a one-to-many operation from the landing zone to all compute nodes and is used during bulk loads. The operation takes blocks of rows as input and distributes them across the compute nodes in a round-robin fashion in row-block units.
- *Master_Move*. This is a one-to-many data movement operation that takes an input table or result set from the control node and replicates it to all compute nodes.
- *Trim_Move*. This is a intra-node data movement operation. It takes a replicated table as input and transforms it into a distributed table by trimming the rows of the replicated table in-place that do not belong to a particular distribution. Since all the nodes have a copy of the replicated table the idea is that each compute node keeps the rows that belong to the distributions on the node.

- *Replicate_Move*. This is a one-to-many data movement operation from one compute node to n-1 compute nodes. It takes a replicated table in a compute node as input and replicates it to all remaining compute nodes.
- *Replicate_Load.* This is a one-to-many operation from the landing zone to all compute nodes and is used during bulk loads. The operation takes blocks of rows as input and replicates them across all compute nodes.

DMS participates in all bulk-load ETL scenarios from the landing zone. The DMS service mediates between PDW and SQL Service Integration Services via a custom PDW destination adapter when loading data from outside the appliance. All collaborating DMS service instances running in all control, compute and landing zone nodes of the appliance are connected via custom data channels built on the high-performance Infiniband network.

PDW Agent. The PDW agent is a windows service that runs on every node in the appliance. This service actively polls Performance Counters, SNMP and Windows WMI data with a defined interval in order to collect health information across all key components in both the hardware and software stack. The data is delivered to a persistent store on the control node to ensure data persistence and fault tolerance. To allow for consistency across vendor topologies, SQL Server PDW created an abstracted health model that defines the nodes, components and properties to monitor and how to access them. Therefore, each vendor specific appliance will include its own health definition file, on each node, which is used to bootstrap the service. The rules defined in the health file also include defining alerts which administrators can act on. The alerts are triggered either when the defined "status" property changes state, or when a property's value violates a threshold rule. Alerts will include information like: create time, name, severity of the alert, as well as descriptive knowledge about the problem, why it is occurring and what further actions are required by the administrator. The alerts and the component health data are available as Dynamic Management Views (DMVs) to users with correct permissions.

***Admin Console*.** The browser-based Admin Console presents a dashboard with a summary of the appliance health status, and it also allows the real-time monitoring of query execution status. Additionally the admin console allows users to view and drill into sessions, queries, loads, backup/restores, locks, application errors, and hardware and software health across the appliance. The admin console is built on a set of dynamic management views (DMVs) which aggregate multiple DMVs and simplifies the access of that information. It also provides a near-real-time view of performance counters from across the appliance in a chart-style interface.

***Deployment Services*.** Using deployment services appliance administrators can execute a single command to apply software patches on appliance nodes or install General Distribution Releases (GDRs) leveraging Windows Supportability Update Service (WSUS). The deployment service uses Windows High Performance Computing (HPC) as a job execution engine to manage the execution of scripts and actions on appliance nodes. HPC is not used during the execution of queries or processing of user data.

3 Layered Architecture and Data Distribution Policies

The SQL Server PDW database system architecture is layered in a way that it separates local from distributed data management concerns. Each compute node runs an instance of SQL Server which manages a local user database partition. Each database partition on a compute node maintains its local metadata (i.e., database, tables, and indexes). Each SQL Server instance running on a compute node is independent and autonomous from all other sibling instances running on other compute and control nodes. Each SQL Server instance is autonomous in the sense that it processes requests received from the PDW Engine which are scoped to run only with the local user data partition.

The PDW Engine is responsible for coordinating the distributed data management concerns. It takes care of maintaining a distributed view of the metadata, sometimes called global metadata. This means that for every database, table, index, or stored procedure created by a user, there is a corresponding entry in the global metadata. Statement compilation is performed in the PDW Engine from a distributed perspective and it produces a sequence of execution tasks, each of which execute on the compute nodes. The communication between the control node and the local SQL Server in each compute node is through standard SQL.

The DMS service serves as the bridge between the distributed data manager (i.e., the PDW Engine) and the local data managers (the SQL Server instances running on each compute node). The DMS service acts as a distributed statement execution engine using the DMS operator primitives described earlier.

SQL Server PDW also offers a built-in collection of data distribution policies. Tables can be declared to be *distributed* based on a hash partition scheme on one of its columns, or they can be declared as *replicated* tables. Usually, large fact tables are distributed while smaller dimension tables are replicated. Figure 3 illustrates an example of a distributed and a replicated table.

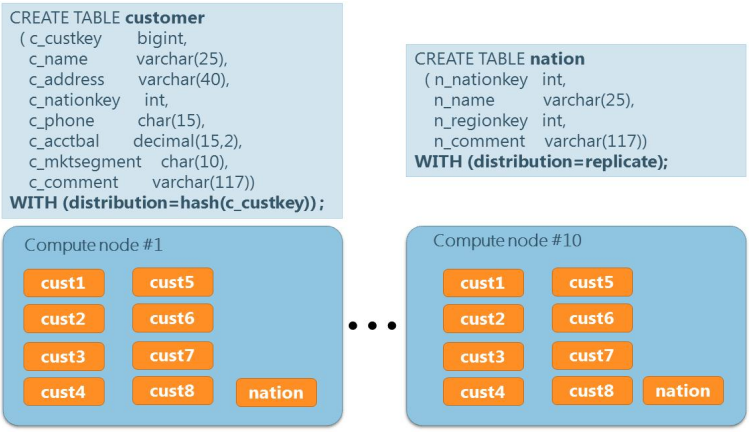


Fig. 3. SQL Server PDW table distribution policies

Table **Customer** is distributed on its **c_custkey** column. A *distributed* table is hash partitioned into 8*n* partitions, also called distributions, where *n* is the number of compute nodes in the appliance. Each compute node holds 8 local distributions of the table. Assuming an appliance with 10 compute nodes, the **Customer** table will be partitioned into 80 distributions, 8 distributions per compute node. Table **Nation** is replicated on all compute nodes.

A key to unlimited scalability in a distributed MPP database system is the ability to maximize locality of reference during query execution. Replicated tables allow joins between a distributed fact table and the replicated table to be collocated. Avoiding or minimizing data movement increases query execution performance in a distributed system.

4 Query Processing

Query processing in SQL Server PDW is also built in a layered approach as outlined in the previous section and can be described in terms of the following three phases.

Distributed Query Compilation. In the *distributed query compilation* phase a query is compiled globally in the PDW Engine process on the control node using global metadata. This phase involves parsing, security access validation, query simplification and an initial logical plan exploration. This phase performs algebraic *query simplification* which involves column reduction, predicate push-down, and subquery unnesting [4]. After query simplification, the optimizer performs a *logical space exploration* which involves join re-ordering and local-global aggregation assuming the data is not distributed. After space exploration, *distributed query optimization* is performed. This query optimization phase identifies interesting distributed properties such as partitioning, and table collocation; injects DMS data movement primitives (e.g., shuffle-move, partition-move) as described in the DMS paragraph within Section 2.2; performs costing of alternatives, pruning, and selection of the lowest estimated distributed plan. The final part of this phase is the generation of a distributed query execution plan which involves a sequence of SQL and data movement operations (also called *query steps*) to be executed among the compute nodes of the appliance.

Local Query Processing. Each step in a distributed query execution plan involves a SQL statement (DDL or DML) or a data movement primitive. Each of these steps is processed by a local SQL Server instance on each compute node. Each of these statements is processed by the SQL Server instance in a standard way. The result set from the execution of such statements serves as the input to a data movement operation which will then combine the result with other results being produced by other compute nodes or returned to the control node.

Distributed Query Execution. Once a distributed query execution plan is produced the PDW Engine coordinates the execution of each of the query steps in the plan until all steps complete and the results are returned to the client. Typically the final step of a distributed plan is executed on the control node and it involves the execution of a partition-move operation which combines partial results produced by the compute

nodes (e.g., global aggregation, final join assembly from collocated join fragments, and final merge in a distributed sort-merge) into the final result.

Query Processing Example

Consider the tables: **Customer**, **Orders**, and **LineItem** in the TPC-H schema distributed on columns **c_custkey**, **o_orderkey**, and **l_orderkey**, respectively. Consider the query *Find the ten largest "building" orders shipped since March 5, 2010* written as follows:

```
SELECT TOP 10 L_ORDERKEY, SUM (L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
        O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_MKTSEGMENT = 'BUILDING' AND C_CUSTKEY = O_CUSTKEY
   AND L_ORDERKEY = O_ORDERKEY AND O_ORDERDATE < '2010-03-05'
   AND L_SHIPDATE > '2010-03-05'
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
```

The distributed query compilation phase produces the following optimal plan:

Step 1: Create a temp table at control node to hold the final partial results.

```
CREATE TABLE [tempdb].[dbo].[Q_[TEMP_ID_664]]
        ( [l_orderkey] BIGINT, [REVENUE] DECIMAL(38, 4),
          [o_orderdate] DATE, [o_shippriority] INTEGER );
```

Step 2: Create a distributed temp table across all compute-nodes hashed on o-custkey. This will hold the temporary result of intermediate co-located join between orders and lineitem.

```
CREATE TABLE [tempdb].[dbo].[Q_[TEMP_ID_665]_[PARTITION_ID]]
        ( [l_orderkey] BIGINT,  [l_extendedprice] DECIMAL(15, 2),
          [l_discount] DECIMAL (15, 2), [o_orderdate] DATE,
          [o_shippriority] INTEGER, [o_custkey] BIGINT,
          [o_orderkey] BIGINT)
WITH (DISTRIBUTION = HASH ([o_custkey]) );
```

Step 3: SHUFFLE_MOVE the result of the join between orders and lineitem and store the result into the distributed temp table.

```
SELECT [l_orderkey], [l_extendedprice], [l_discount], [o_orderdate], [o_shippriority],
        [o_custkey], [o_orderkey]
FROM [dwsys].[dbo].[orders] JOIN [dwsys].[dbo].[lineitem]
ON ([l_orderkey] = [o_orderkey])
WHERE ([o_orderdate] < '2010-03-05'
        AND [o_orderdate] >= '2010-09-15 00:00:00.000')
        INTO Q_[TEMP_ID_665]_[PARTITION_ID]
SHUFFLE ON (o_custkey);
```

Step 4: PARTITION_MOVE the join between the distributed temp table and customer and the local aggregation of revenue. Store the local aggregations on the temp table at the control node.

```
SELECT [l_orderkey], sum(([l_extendedprice] * (1 - [l_discount]))) AS REVENUE,
        [o_orderdate], [o_shippriority]
FROM [dwsys].[dbo].[customer] JOIN tempdb.Q_[TEMP_ID_665]_[PARTITION_ID]
      ON ([c_custkey] = [o_custkey])
WHERE [c_mktsegment] = 'BUILDING'
GROUP BY [l_orderkey],  [o_orderdate], [o_shippriority]
INTO Q_[TEMP_ID_664];
```

Step 5: RETURN the global aggregation and final result to client.

```
SELECT TOP 10 [l_orderkey], sum([REVENUE]) AS REVENUE, [o_orderdate], [o_shippriority]
FROM tempdb.Q_[TEMP_ID_664]
GROUP BY [l_orderkey],  [o_orderdate],  [o_shippriority]
ORDER BY [REVENUE] DESC, [o_orderdate] ;
```

Step 6: Final clean up – drop all temp tables.

```
DROP TABLE tempdb.Q_[TEMP_ID_665]_[PARTITION_ID];
DROP TABLE tempdb.Q_[TEMP_ID_664];
```

5 Additional Functionality

A commercial distributed database system like SQL Server PDW includes many critical components beyond query processing to achieve its function. This section briefly describes a few of these components.

Transaction Management. SQL Server PDW supports local and distributed transactions. Each SQL Server instance running on a compute or control node implements local transaction management. This means full ACID properties and isolation levels. Distributed transaction management is carried out by the PDW Engine using the Microsoft Distributed Transaction Coordinator (DTC) which is a components built-into Windows Server. Statements that require ACID properties across all nodes of the appliance are scoped under a distributed transaction. The default isolation level for metadata is read committed. The default isolation level for user data is read uncommitted which is typical of DW workloads that involve a large proportion of concurrent queries mixed with a small proportion of bulk inserts and rare in-place updates.

Backup and Restore. SQL Server PDW provides an appliance-wide database backup and restore capability which builds on the backup and restore functionality of the local SQL Servers running on each compute node. An appliance-wide backup is broken into *n* parallel backups, one per compute node. Each backup image is stored on the backup node and organized within a common location. SQL Server PDW allows these backups to be offloaded to an alternate device using an enterprise backup

solution. Restores perform the reverse operation and are also executed in parallel. There is support for full and differential backups.

Fault Tolerance. SQL Server PDW is a fault-tolerant system. There is redundancy built-into nearly all hardware and software components. At the storage level, user data is stored redundantly using a RAID1 mirroring. There is one spare compute node per every ten compute nodes. All control and management nodes are configured in pairs of active/passive servers. The network switches and cards have dual paths. At the software level, PDW builds on the Microsoft Failover Clusters (FC) component built into Windows Server. FC provides hardware and software failure detection mechanisms as well as failover policy implementation.

Appliance Health. SQL Server PDW includes a comprehensive collection of hardware and software instrumentation to monitor the health of the system. All nodes of the appliance run an instance of the PDW Agent service. This agent collects all relevant events in the system and rolls them up into logical units representing servers, storage, network, etc. These events are collected and aggregated up to the PDW Agent running on the control node where the global health of the appliance is exposed via a set of dynamic maintenance views (DMVs). Management tools like the Admin Console query these DMVs to source the data displayed in the appliance health dashboard. In addition, a Systems Center Operations Manager (SCOM) pack is available for PDW appliances. This allows administrators to use SCOM to implement automated monitoring and alerting of PDW appliances.

PDW Appliance Validator. As mentioned earlier in this paper, SQL Server PDW involves hundreds of hardware and software components. When the appliance is not functioning properly it is important to quickly determine whether the issue is due to a hardware or software malfunction. The PDW appliance validator is a set of utilities that check the state of all critical HW and SW components in the system. It aggregates the results of the checks across the entire appliance and reports the state in a dashboard. The appliance validator is run before and after critical patching or upgrade servicing tasks to ensure the appliance is in good state before and after the maintenance.

6 Summary and Conclusions

This paper presented an overview of the SQL Server PDW database system, a scale-out, shared-nothing, MPP system packaged as an appliance and focused on delivering high-performance for DW workloads. We presented and overview of the appliance including its main hardware and software components. The software system has been built using a layered approach that separates distributed from local data management concerns. We presented an overview of the distributed query processor and some of the key components of the system. Future versions of the PDW system will incorporate column-store technology, which ships in the SQL Server 2012 SMP release, as well as deeper integration with BI analytics and big-data technologies such as Hadoop.

Acknowledgments. The layered approach to building a distributed, MPP DBMS and the early appliance design was conceived at DatAllegro by Stewart Frost, Dave Salch, Paul Dyke and Matt Peebles. Building and shipping the SQL Server PDW product is a large team effort involving dozens of engineers across the SQL Server PDW product unit at Aliso Viejo, CA; the Serbia Development Center; the Jim Gray Systems Laboratory at Madison, WI; and the Database Systems Group at Redmond, WA. We are grateful to all engineers at these centers who have contributed to the development of SQL Server PDW.

References

1. Astrahan, M., et al.: System R: Relational Approach to Database Management. ACM TODS 1(2) (1976)
2. Stonebraker, M., et al.: The Design and Implementation of INGRES. ACM TODS 1(3) (1976)
3. Stonebraker, M., et al.: The End of an Architectural Era (It's Time for a Complete Rewrite). In: Proc. VLDB (2007)
4. Agarwal, S., et al.: Microsoft SQL Server. In: Silberschatz, A., Korth, H.F., Sudarshan, S. (eds.) Database System Concepts, 6th edn., ch. 30. McGraw-Hill (January 2010)

Relax and Let the Database Do the Partitioning Online

Alekh Jindal and Jens Dittrich

Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

Abstract. Vertical and Horizontal partitions allow database administrators (DBAs) to considerably improve the performance of business intelligence applications. However, finding and defining suitable horizontal and vertical partitions is a daunting task even for experienced DBAs. This is because the DBA has to understand the physical query execution plans for each query in the workload very well to make appropriate design decisions. To facilitate this process several algorithms and advisory tools have been developed over the past years. These tools, however, still keep the DBA in the loop. This means, the physical design cannot be changed without human intervention. This is problematic in situations where a skilled DBA is either not available or the workload changes over time, e.g. due to new DB applications, changed hardware, an increasing dataset size, or bursts in the query workload. In this paper, we present AutoStore: a self-tuning data store which rather than keeping the DBA in the loop, monitors the current workload and partitions the data automatically at checkpoint time intervals — without human intervention. This allows AutoStore to gradually adapt the partitions to best fit the observed query workload. In contrast to previous work, we express partitioning as a One-Dimensional Partitioning Problem (1DPP), with Horizontal (HPP) and Vertical Partitioning Problem (VPP) being just two variants of it. We provide an efficient O^2P (One-dimensional Online Partitioning) algorithm to solve 1DPP. O^2P is faster than the specialized affinity-based VPP algorithm by more than two orders of magnitude, and yet it does not loose much on partitioning quality. AutoStore is a part of the OctopusDB vision of a One Size Fits All Database System [13]. Our experimental results on TPC-H datasets show that AutoStore outperforms row and column layouts by up to a factor of 2.

Keywords: changing workload, online partitioning, self-tuning.

1 Introduction

Physical database designs have been researched heavily in the past [23, 3, 2, 5, 6, 31, 7, 22, 26, 12]. As a consequence, nowadays, most DBMSs offer design advisory tools [1, 32, 30, 4]. These tools help DBAs in defining indexes, e.g. [8], as well as horizontal and/or vertical partitions, e.g. [3, 15]. The idea of these tools is to analyze the workload at a given point in time and suggest different physical designs. These suggestions are computed by a what-if analysis. What-if

M. Castellanos, U. Dayal, and W. Lehner (Eds.): BIRTE 2011, LNBIP 126, pp. 65–80, 2012.
© Springer-Verlag Berlin Heidelberg 2012

analysis explores the possible physical designs. However, just finding the right set of partitions is NP-hard [29]. Therefore the search space must be pruned using suitable heuristics, i.e. typically some greedy-strategy [2]. The cost of each candidate configuration is then estimated using an existing cost-based optimizer, i.e. the optimizer is tricked into believing that the candidate configuration already exists. Eventually, a suitable partitioning strategy is proposed to the DBA who then has to re-partition the existing database accordingly.

1.1 Problems with Offline Partitioning

The biggest problem with this approach is that it is an *offline* process. The DBA will only reconsider the current physical design at certain points in time. This is problematic. Assume the workload changes over time, e.g. changes in the workload due to new database applications, an increasing dataset size, or an increasing number of queries. In these situations the existing partitioning strategies should be revisited to improve query times. In the current *offline approach* however, the partitioning strategies will only be changed if a human — the DBA — triggers an advisory tool with the most recent query logs and eventually decides to repartition the data. This means, the vertical and horizontal partitioning strategies are carved in stone until the DBA changes them eventually. Furthermore, current advisory tools attempt to find near optimal partitioning strategies, which is very expensive. This is especially problematic if the database system has to handle bursts and peaks. For instance consider (i) a ticket system selling 10 million Rolling Stones tickets within three days; (ii) an online store such as Amazon selling much higher volumes before christmas; or (iii) an OLAP system having to cope with new query patterns. In these types of applications it is not acceptable for users to wait for the DBA and the advisory tool to reconfigure the system. If the system stalls due to a peak workload, the application provider may loose a lot of money.

1.2 Research Goals and Challenges

Our **goal** is to research a database store that decides on the suitable partitioning strategy *automatically*, i.e. without any human intervention. As the search space of possible partitions is huge [29], it is clear from the beginning that an *optimal* automatic partitioning is not always feasible. However, we believe that an automatic (or: *online*) partitioning will in most cases be much better than the one suggested by even a skilled DBA — similarly the physical query execution plans being in most cases better than hand-crafted plans. The risk of not reaching optimality is similar to the risk of adaptive indexing [14] and cracking [19]. However, the possible gains of such an approach may be similarly tremendous.

This leads to interesting **research challenges**:

(1.) There exist a plethora of state-of-the-art offline algorithms, e.g. [23, 24, 2], for suggesting suitable vertical and horizontal partitions. However, given the huge search space, it turns out that their runtime complexity is unacceptably high to be applicable in an online setting where the available time to decide on a new partitioning is rather limited. Therefore we must develop new algorithms.

(2.) We need algorithms that decide on data partitioning automatically and while the database is running, i.e. take decisions to repartition the data.
(3.) Any automatic repartitioning must not block or stall the database and/or access to entire tables, a problem more likely in archival disk-based databases.

1.3 Contributions

In this paper, we present AutoStore, a fully automatic database store, to solve these challenges. To the best of our knowledge, this work is the first to solve the database partitioning problem with a fully automatic *online approach*. The contributions of this paper are:

(1.) We express partitioning as general One-Dimensional Partitioning Problem (1DPP), with Vertical (VPP) and the Horizontal Partitioning Problem (HPP) as subproblems of it. Both subproblems may be solved by solving 1DPP (Section 2).
(2.) We present AutoStore, an online self-tuned database store that is a step towards implementing the OctopusDB vision [13, 21]. The core components of AutoStore are: dynamic workload monitor, partitioning unit clusterer, partitioning analyzer, and partitioning optimizer (Section 3).
(3.) We present an online database partitioning algorithm O^2P (One-dimensional Online Partitioning) to solve 1DPP (Section 4).
(4.) We show an extensive evaluation of our algorithm over TPC-H and SSB benchmarks. We present experimental results from mixed OLTP/OLAP workloads over a main-memory and a BerkeleyDB implementation (Section 5).

2 Vertical and Horizontal Partitioning

2.1 Preliminaries

Typically, users partition databases horizontally based on data value ranges, hashes, or lists. This is because data values are comparable across a column. However, this is not true for data values across a row. Therefore, sophisticated partitioning methods have been developed for VPP. In this section we will revisit the basics of VPP. This also serves as the ground work for our 1DPP.

Naïve Approach. Database researchers pointed out the heuristic [16] and NP-hard [29] nature of partitioning problem pretty early. The number of ways to partition vertically, for x attributes, is given by bell number $B(x)$. The naïve approach to find the optimal solution is to enumerate all bell numbers. The complexity of the naïve approach is $O(x^x)$, making it infeasible for large databases.

Affinity Based Approach. The naïve approach considers all possible partitions, even the ones having attributes which are *never* accessed together. To address this, *attribute affinity* was introduced as a measure of pairwise attribute similarity [18, 23, 10, 24, 11]. The core idea of affinity based partitioning is to compute affinities between every pair of attributes and then to cluster them

such that high affinity pairs are as close in neighborhood as possible. To compute affinity between different attributes, we need to know their access patterns. A *usage function* $U(q, a)$ denotes whether or not query q references attribute a. $U(q, a) = 1$ if q references a and 0 otherwise. For example, in TPC-H Lineitem table, $U(Q_1, \text{PartKey}) = 1$ as Query 1 references attribute PartKey. The usage function may also be extended to incorporate query weights, reflecting the importance levels or relative frequencies of queries. To measure the affinity between two attributes a_i and a_j, the *affinity function* $A(a_i, a_j)$ simply counts their co-occurrences in the query workload, i.e. $A(a_i, a_j) = \sum_q U(q, a_i) \cdot U(q, a_j)$. For instance, in Lineitem table, $A(\text{PartKey}, \text{SuppKey}) = 5$, as attributes PartKey and SuppKey co-occur in five queries. The affinity function produces a 2D affinity matrix between every pair of attributes. The goal now is to cluster the matrix such that the cells having similar affinity values are placed close together in the matrix. Every order of rows and columns in the matrix gives a new *ordering of attributes* ($\preceq$). For example, consider the following affinity matrices for PartKey, SuppKey, and Quantity attributes in TPC-H Lineitem table. The left matrix represents an attribute ordering PartKey $\preceq$ SuppKey $\preceq$ Quantity, whereas the right matrix represents ordering PartKey$\preceq$Quantity$\preceq$SuppKey. Given attribute ordering $\preceq$, an *affinity measure* $M(\preceq)$ measures the quality of the affinity clustering as $M(\preceq) = \sum_{i=1}^{x} \sum_{j=1}^{x} A(a_i, a_j)[A(a_i, a_{j-1}) + A(a_i, a_{j+1})]$. It holds that $A(a_0, a_j) = A(a_i, a_0) = A(a_{x+1}, a_j) = A(a_i, a_{x+1}) = 0$. For the left matrix above $M(\preceq) = 404$ and for the right matrix $M(\preceq) = 440$. Indeed, the right matrix has better clustering since affinity between attributes PartKey and Quantity (=6) is more than that between PartKey and SuppKey (=5). Thus, the objective of affinity matrix clustering problem now is to maximize the affinity measure. One (greedy) approach is to place the attributes one-by-one such that the *contribution to the affinity measure* at each step is maximized [23]. The contribution to the affinity measure of a new attribute a_k when placed between two already placed attributes a_i and a_j is: $Cont(a_i, a_j, a_k) = 2 \cdot \sum_{z=1}^{n} [A(a_z, a_i) \cdot A(a_z, a_k) + A(a_z, a_k) \cdot A(a_z, a_j) - A(a_z, a_i) \cdot A(a_z, a_j)]$. In this clustering approach, we first place two random attributes; then, in the neighborhood, we place the attribute which maximizes the contribution to the affinity measure. We repeat this process until all attributes are placed.

	PartKey	SuppKey	Quantity
PartKey	8	5	6
SuppKey	5	8	4
Quantity	6	4	9

	PartKey	Quantity	SuppKey
PartKey	8	6	5
Quantity	6	9	4
SuppKey	5	4	8

2.2 Problem Statement

In this section we express HPP and VPP as a general 1DPP. The first step to do so is to identify the smallest indivisible units of storage.

Definition 1. *A **partitioning unit set** $P_u = \{u_1, u_2, .., u_n\}$ is the set of n smallest pieces of data.*

Definition 2. *A **partitioning unit ordering** $\preceq$ defines an order on the partitioning units in P_u.*

Partitioning units could be attributes along the vertical axis or tuples along the horizontal axis. However, partitioning at the tuple level may not make sense due to large number of partitioning units and hence high complexity. Therefore, we usually consider sets of tuples, based on some key, as partitioning units (horizontal partitioning). Similarly, we could also consider groups of columns as partitioning units (vertical partitioning). Below, we introduce some new concepts needed for our one-dimensional partitioning problem statement. First, we express partitioning as a logical partitioning, to be able to use it in an algorithm.

Definition 3. *A **split vector** S is a row vector of (n-1) split lines in ordering $\preceq$, where a split line s_j is defined between partitioning units u_j and u_{j+1} as follows:*

$$s_j = \begin{cases} 1 & \text{if there is split between } u_j \text{ and } u_{j+1} \\ 0 & \text{for no split} \end{cases}.$$

A split vector S captures the logical partitioning over a given dataset. For instance, a split vector S_1=[0,0,0,1,0,1,1] corresponds to a partitioning of $u_1, u_2, u_3, u_4|u_5, u_6|u_7|u_8$. However, in order to estimate costs using a cost-based query optimizer, a split vector still needs to be translated in terms of partitioning units:

Definition 4. *A **partition** $p_{m,r}(S,\preceq)$ is a maximal chunk of adjacent partitioning units from u_m to u_r, such that split lines s_m to s_{r-1} are all* 0.

Definition 5. *A **partitioning scheme** $P(S,\preceq)$ over relation R is a set of disjoint and complete partitions, i.e.*

$$\bigcup_x p_{m_x,r_x}(S,\preceq) = R,$$

$$p_{m_x,r_x}(S,\preceq) \cap p_{m_y,r_y}(S,\preceq) = \phi, \forall x, y \text{ such that } x \neq y.$$

Partitioning scheme expresses the actual arrangement of partitioning units, given a split vector. For instance, for split vector S_1, partition $p_{1,4}(S_1,\preceq)$ is $\{u_1, u_2, u_3, u_4\}$ and partitioning scheme $P(S_1,\preceq)$=$\{p_{1,4}(S_1,\preceq), p_{5,6}(S_1,\preceq), p_7(S_1,\preceq), p_8(S_1,\preceq)\}$. Finally, in order to evaluate partitioning schemes in an online setting, we need to model the online query workload.

Definition 6. *An **Online Workload** W_{t_k} is a stream of queries $\{q_0, ..., q_{t_{k-1}}, q_{t_k}\}$ seen till time t_k, where $t_k > t_{k-1} > ... > 0$.*

Further, let $C_{\text{est.}}(W_{t_k}, P(S,\preceq))$ denote the execution cost of workload W_{t_k} as estimated by a cost-based optimizer. Now, we express our one-dimensional partitioning problem as follows.

One-Dimensional Online Partitioning Problem. *Given an online workload W_{t_k} and partitioning unit ordering $\preceq$, find the split vector S' that minimizes the estimated workload execution cost, i.e.*

$$S' = \underset{S}{\operatorname{argmin}}\, C_{\text{est.}}\Big(W_{t_k}, P(S,\preceq)\Big). \tag{1}$$

The complexity of the above problem depends on the number of partitioning schemes P, which in turn depends on the range of values of split vector S. Note that the one-dimensional partitioning problem has the following sub-problems: (1) *Vertical Partitioning*, if the partitioning unit set P_u is a set of attributes, and (2) *Horizontal Partitioning*, if the partitioning unit set is a set of horizontal ranges.

In the next section we describe our AutoStore system and discuss how it solves the one-dimensional partitioning problem in an online setting.

3 AutoStore

In this section we present AutoStore, an automatically and online partitioned database store. The workflow in AutoStore is as follows: (1) dynamically monitor the query workload and dynamically cluster the partitioning units; (2) analyze the affinity matrix at regular intervals and decide whether or not to create the partitions; and (3) keep data access unaffected while monitoring workload and analyzing partitions. The salient features of AutoStore are as follows:

(1.) *Physical Data Independence.* Instead of exposing physical data partitioning to the user, AutoStore hides these details. This avoids mixing the logical and physical schema. Thus, AutoStore offers better physical data independence, which is not the case in traditional databases.
(2.) *DBA-oblivious Tuning.* The DBA is not involved in triggering the right data partitioning. AutoStore self-tunes its data.
(3.) *Workload Adaptability.* AutoStore monitors the *changes* in query workload and automatically adapts data partitioning to it.
(4.) *Generalized Partitioning.* AutoStore treats partitioning as a 1DPP. Subproblems VPP and HPP are handled equivalently by rotating the table through ninety degrees, i.e. changing the partitioning units from attributes to ranges.
(5.) *Cost, Benefit Optimization.* To decide whether or not to actually partition data, AutoStore considers both the costs as well as the benefits of partitioning.
(6.) *Uninterrupted Query Processing.* AutoStore makes use of our online algorithm O^2P which amortizes the computationally expensive partitioning analysis over several queries. The query processing remains uninterrupted.

Below we discuss four crucial components — workload monitor, partitioning unit clusterer, partitioning analyzer, and partitioning optimizer — which make online self-tuning possible in AutoStore.

Workload Monitor. Online partitioning faces the challenge of creating good partitions for future queries based on the seen ones. One might consider partitioning after every incoming query. However, not only could this be expensive (due to data shuffling), the next incoming query could be entirely different from the previous one. Hence, we maintain a *query window* to capture the workload pattern and have greater confidence over partitioning decisions. Additionally, we slide the query window once it grows to a maximum size, to capture the latest workload trends. We denote a sliding query workload having N queries as $W_{t_k}^N \subseteq W_{t_k}$. After every `CheckpointSize` number of new queries, AutoStore triggers partitioning analysis, i.e., it takes a snapshot of the current query window and the partitioning unit ordering and determines the new partitioning.

Partitioning Unit Clusterer. The partitioning unit clusterer is responsible for re-clustering the affinity matrix after each incoming query. The affinity matrix

clustering algorithm (in Section 2.1) has the following issues: (i) it recomputes all affinity values, and (ii) it reclusters the partitioning units from scratch. We need to adapt it for online partitioning in AutoStore. The core idea is to compute all affinities once and then for each incoming query update only the affinities between referenced partitioning units. Note that the change in each of these affinity values will be 1, due to co-occurrence in the incoming query. For example, consider TPC-H Lineitem table having the affinity matrix shown at left below. Now, for an incoming query accessing PartKey and SuppKey, only the affinities between them are updated (gray cells in the right affinity matrix above). Likewise, we need to re-cluster only the referenced partitioning units. To do this, we keep the first referenced partitioning unit at its original position, and for the i^{th} referenced unit we consider the left and right positions of the $(i-1)$ referenced units already placed. We calculate the net contribution of i^{th} referenced unit to the global affinity measure as: (*Cont* at the new position) – (*Cont* at the current position). We choose the position that offers maximum net contribution to the global affinity measure and repeat the process for all referenced partitioning units. To illustrate, in the right affinity matrix above, we first place PartKey and then consider placing SuppKey to the left (net contribution=48) and right (net contribution=0) of Partkey. Thus, we will place SuppKey to the left of PartKey.

	PartKey	Quantity	SuppKey
PartKey	8	6	5
Quantity	6	9	4
SuppKey	5	4	8

	PartKey	Quantity	SuppKey
PartKey	9	6	6
Quantity	6	9	4
SuppKey	6	4	9

Partitioning Analyzer. The partitioning analyzer of AutoStore analyzes partitioning every time `CheckpointSize` number of queries are added by the workload monitor. The job of the partitioning analyzer is to take a snapshot of the query window as input, enumerate and analyze the partitioning candidates, and emit the best partitioning as output. In the brute force enumeration approach, we consider all possible values (0 or 1), for each split line in the split vector S of Equation 1. We then pick the split vector which produces the lowest estimated workload execution cost $C_{\text{est.}}(W^N_{t_k}, P(S,\preceq))$. Each split vector gives rise to a different candidate partitioning scheme. The size of the set of candidate partitioning schemes is $2^{n-p-q-1}$. In Section 4 we show how the O^{2}P algorithm significantly improves partitioning analysis in an online setting.

Partitioning Optimizer. Given the partitioning scheme P' produced by the partitioning analyzer, the partitioning optimizer decides whether or not to transform the current partitioning scheme P to P'. The partitioning optimizer considers the expected costs of transforming the partitioning scheme as well as the expected benefits from it. We discuss these considerations below.

Cost Model. We use a cost model for full table and index scan operations over one-dimensional partitioned tables. To calculate the partitioning costs, we first find the partitions in P which are no longer present in P': $P_{\text{diff}} = P \setminus P'$. Now, to transform from P to P' we simply have to read each of the partitions in P_{diff} and store it back in the required partitions in P'. For instance, the transformation

cost for vertical partitioning can be estimated as twice the scan costs of partitions in P. From such a transformation cost model, the worst case transformation cost is equal to twice the full table scan, whereas the best case transformation cost is twice the scan cost of the smallest partitioning unit.

Benefit Model. Same as we compute the cost of partitioning, we also need the benefit of partitioning in order to make a decision. We model partitioning benefit as the difference in the cost of executing the query window on the current and the new partitioning, i.e. $B_{\text{transform}} = C_{\text{est.}}(W_{t_k}^N, P(S,\preceq)) - C_{\text{est.}}(W_{t_k}^N, P'(S,\preceq))$.

Partitioning Decision. For each transformation made, we would have recurring benefits over all similar workloads. Hence, we need to factor in the expected frequency of the query window. This could be either explicitly provided by the user or modeled by the system. For instance, an exponential decaying model with shape parameter y: Workload Frequency$(f) = \frac{1}{1-y^{-\texttt{MaxWindowSize}}}$ gives higher frequency to smaller query windows. AutoStore creates the new partitioning only if the total recurring partitioning benefit ($\texttt{pBenefit} = f \cdot B_{\text{transform}}$) is expected to be higher than the partitioning cost ($\texttt{pCost} = C_{\text{transform}}$).

Partitioning Transformation/Repartitioning. Repartitioning data from P to P' poses interesting algorithmic research challenges. As stated before the overall goal should be to minimize transformation costs. In addition, the database or even single tables must not be stalled, i.e. by halting incoming queries. Fortunately, these problems may be solved. In a *read-only* system any table or horizontal partition may be transformed in the background, i.e. we transform P to P' and route all incoming queries to P. Only if the transformation is finished, we atomically switch to P'. For *updates*, this process can be enriched by keeping a differential file or log L of the updates that are arriving while the tranformation is running. Any incoming query may then be computed by considering P and L. If the tranformation is finished, we may eventuall decide to merge P' with P. The right strategy for doing this is not necessarily to merge immediately. A similar discussion as for LSM trees [25] and exponential files [20] applies. For repartitioning vertical layouts there are other interesting challenges. None of them would require us to stall incoming queries or halt the database. We are planning to evaluate these algorithms in a separate study.

4 O²P Algorithm

The partitioning analyzer in Section 3 described the brute force approach of enumerating all possible values of split vector S in the one-dimensional partitioning problem. This approach has exponential complexity and hence is not desirable in an online setting. In this section, we present an online algorithm O^{2}P (One-dimensional Online Partitioning) which solves 1DPP in an online setting. O^{2}P does not produce the optimal partitioning solution. Instead, it uses a number of techniques to come up with greedy solution. The greedy solution is not only dynamically adapted to workload changes, it also does not loose much on partitioning quality as well. Below we highlight the major features in O^{2}P:

(1.) Partitioning Unit Pruning. Several partitioning units are never referenced by any of the queries in the query window. For instance, `RetailPrice` is not referenced in TPC-H `Part` table. Due to one-dimensional clustering, such partitioning units are expected to be in the beginning or the end of the partitioning unit ordering. Therefore, O^{2}P prunes them into a separate partition right away. As an example, consider 3 leading and 2 trailing partitioning units, out of total 10 partitioning units, to be not referenced. Then, the following split lines are determined: $s_1 = s_2 = 0$, $s_3 = 1$, $s_9 = 0$, $s_8 = 1$.
(2.) Greedy Split Lines. Instead of enumerating over all possible split line combinations — as in the brute force — O^{2}P greedily sets the best possible split line, one at a time. O^{2}P starts with a split vector having all split lines as 0 and at each iteration it sets (to 1) only one split line. To determine which split line to set, O^{2}P considers all split lines unset so far, and picks the one giving the lowest workload execution cost, i.e. the $(i+1)^{th}$ split line to be set is given by: $s_{i+1} = \text{argmin}_{s \in \text{unset}(S_i)}\ C_{\text{est.}}(W^N_{t_k}, P(S_i + U(s), \preceq))$, where $U(s)$ is a unit vector having only split line s as set; corresponding split vector is: $S_{i+1} = S_i + U(s)$.
(3.) Dynamic Programming. Observe that the partitions not affected in the previous iteration of greedy splitting will have the same best split line in the current iteration. For example, consider an ordering of partitioning units with binary partitioning: $u_1, u_2, u_3, u_4 | u_5, u_6, u_7, u_8$. The corresponding split vector is: $[0, 0, 0, 1, 0, 0, 0]$ with only split line s_4 set to 1 and all other split lines set to 0. Now, we consider all unset split lines for partitioning. Suppose s_2 and s_6 are the best split lines in the left and right partitions respectively and amongst them s_2 is the better split line. In next iteration, we already know that s_6 is the best split line in the right partition and only need to evaluate s_1 and s_3 in the left partition again. To exploit this O^{2}P maintains the best split line in each partition and reevaluates split lines only in partitions which are further split. Since it performs only one split at a time (greedy), it only needs to reevaluate the split lines in the most recently split partition. Algorithm 1 shows the dynamic programming based enumeration in O^{2}P. First, O^{2}P finds the best split line and its corresponding cost in: the left and right parts of the last partition, and all previous partitions (Lines 1–6). If no valid split line is found then O^{2}P returns (Lines 7–9). Otherwise, it compares these three split lines (Line 10), chooses the one having lowest costs (Lines 11–35), and repeats the process (Line 36).

Theorem 1. O^{2}P *produces the correct greedy result.*

Theorem 2. *If consecutive splits reduce the partitioning units by z elements, then the number of iterations in* O^{2}P *is* $\left\lceil \frac{(n-3)(n-z-1)}{2z} + 2n - 3 \right\rceil$.

Lemma 1. *Worst case complexity of* O^{2}P *is* $O(n^2)$.

Lemma 2. *Best case complexity of* O^{2}P *is* $O(n)$[1].

[1] All proofs at:
`http://infosys.cs.uni-saarland.de/publications/TRAutoStore.pdf`

Algorithm 1. `dynamicEnumerate`

```
   Input  : S, left, right, PrevPartitions
   Output: Enumerate over possible split vectors
 1 SplitLine sLeft = BestSplitLine(S,left);
 2 Cost minCostLeft = BestSplitLineCost(S,left);
 3 SplitLine sRight = BestSplitLine(S,right);
 4 Cost minCostRight = BestSplitLineCost(S,right);
 5 SplitLine sPrev = BestSplitLine(S,PrevPartitions);
 6 Cost minCostPrev = BestSplitLineCost(S,PrevPartitions);
 7 if invalid(sLeft) and invalid(sRight) and invalid(sPrev) then
 8 |   return;
 9 end
10 Cost minCost = min(minCostLeft, minCostRight, minCostPrev);
11 if minCost == minCostLeft then
12 |   SetSplitLine(S, sLeft);
13 |   if sRight > 0 then
14 |   |   AddPartition(right, sRight, minCostRight);
15 |   end
16 |   right = sLeft+1;
17 else if minCost == minCostRight then
18 |   SetSplitLine(S, sRight);
19 |   if sLeft > 0 then
20 |   |   AddPartition(left, sLeft, minCostLeft);
21 |   end
22 |   left = right;
23 |   right = sRight+1;
24 else
25 |   SetSplitLine(S, sPrev);
26 |   if sRight > 0 then
27 |   |   AddPartition(right, sRight, minCostRight);
28 |   end
29 |   if sLeft > 0 then
30 |   |   AddPartition(left, sLeft, minCostLeft);
31 |   end
32 |   RemovePartition(sPrev);
33 |   left = pPrev.start();
34 |   right = sPrev+1;
35 end
36 dynamicEnumerate(S, left, right, PrevPartitions);
```

(4.) Amortized Partitioning Analysis. O^{2}P computes the partitioning lazily over the course of several queries, i.e. it performs a subset of iterations each time AutoStore triggers the partitioning analyzer. Thus, O^{2}P amortizes the cost of computing the partitioning scheme over several queries. This makes sense, because otherwise we may end up spending a large number of CPU cycles, and blocking query execution, even though the partitioning may not be actually done (due to cost-benefit considerations). The partitioning analyzer returns the best split vector only when all iterations in the current analysis are done.

(5.) Multi-threaded Analysis. Since our partitioning analysis works on a window snapshot of the workload, O^{2}P can also delegate it to a separate secondary thread while the normal query processing continues in the primary thread. This approach completely separates query processing from partitioning analysis.

5 Experiments

The goal of our experiments is three-fold: (1) to evaluate the partitioning analysis in AutoStore on realistic TPC-H and SSB workloads, (2) to compare the

query performance of a main-memory based implementation of AutoStore with No and Full Vertical Partitioning, and (3) to evaluate the performance of AutoStore on a real system: BerkeleyDB. We present each of these in the following. All experiments were executed on a large computing node having Intel Xeon 2.4GHz CPU with 64GB of main memory, and running on Ubuntu 10.10 operating system.

The algorithms of Navathe et. al. [23] and Hankins et. al. [17] have similar complexity. Therefore, we label them as NV/HC. To compare and obtain a cost analysis of different components in O^{2}P, we switch them on incrementally. Thus, we have five different variants of O^{2}P: (i) only partitioning unit pruning (O^{2}Pp), (ii) pruning+greedy (O^{2}Ppg), (iii) pruning+greedy+dynamic (O^{2}Ppgd), (iv) pruning+greedy+dynamic+amortized (O^{2}Ppgda), and (v) pruning +greedy+dynamic+multi-threaded (O^{2}Ppgdm).

5.1 Evaluating Partitioning Analyzer

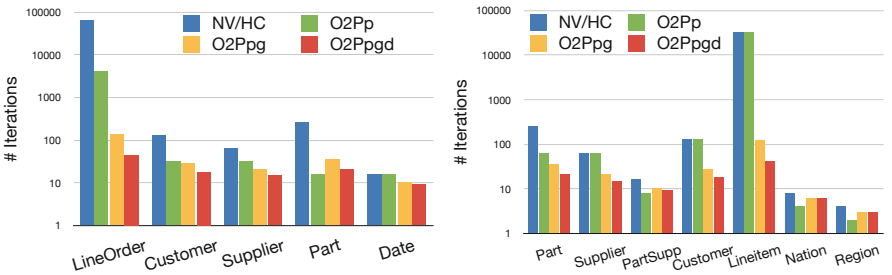
We now evaluate O^{2}P on multiple benchmark datasets and workloads. Figure 1(a) shows the number of iterations in different variants of O^{2}P for different tables in Star Schema Benchmark (SSB). We can see that O^{2}Pp indeed improves over NV/HC on this realistic workload. O^{2}Ppg and O^{2}Ppgd are even better. Figure 1(b) shows the iterations in different variants of O^{2}P for TPC-H dataset. For Lineitem table, O^{2}Ppgd has just 42 iterations compared to 32,768 iterations in NV/HC. O^{2}Ppgda and O^{2}Ppgdm have the same number of iterations as O^{2}Ppgd, hence we do not show them in the figure.

Next, we evaluate the actual running time of different O^{2}P variants while varying the read-only workload. We vary the 100-query workload from OLTP style (1% tuple selectivity, 75-100% attribute selectivity) to OLAP style (10% tuple selectivity, 1-25% attribute selectivity) access patterns. We run this experiment over Lineitem (Figure 2(a)) and Customer tables (Figure 2(b)). We observe that on Lineitem O^{2}Ppgd outperforms NV/HC by up to two orders of maginitude.

Now let us analyze the *quality* of partitioning produced by O^{2}P. We define the quality of partitioning produced by an algorithm as the ratio of the expected query costs of optimal partitioning and the expected query costs of partitioning produced by the algorithm. The table below shows the quality and the number of iterations for optimal, NV, and O^{2}P partitioning over mixed OLTP-OLAP workload. We can see that O^{2}P significantly reduces the number of iterations, without loosing much on partitioning quality.

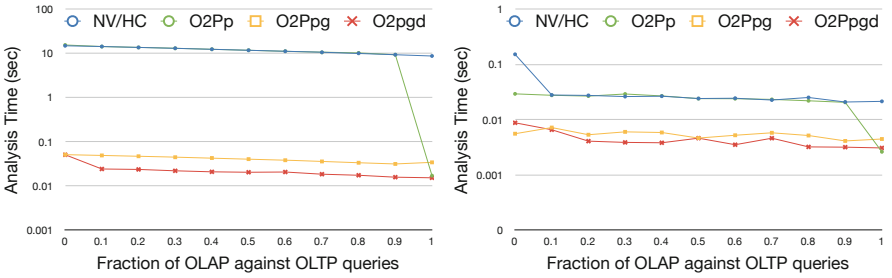
	Customer			Lineitem		
	Optimal	Navathe	O2P	Optimal	Navathe	O2P
Quality	100%	99.29%	92.76%	100%	97.45%	95.80%
Iterations	100%	14.60%	2.28%	100%	2.42%	0.14%

Finally, we evaluate the scalability of O^{2}P when increasing workload size. We vary the workload size from 1 to 10,000 queries consisting of equal number of



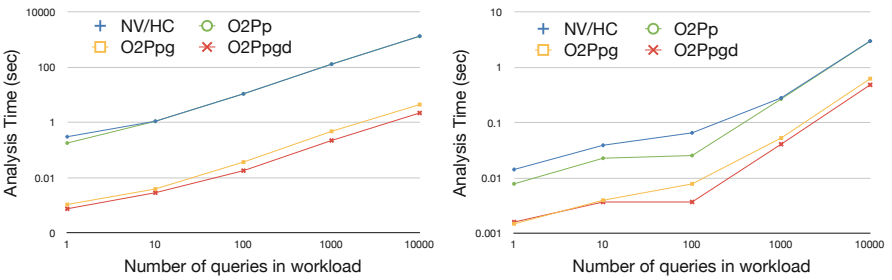
(a) SSB Benchmark (b) TPC-H Benchmark

Fig. 1. Number of iterations in different algorithms over SSB and TPC-H benchmarks on different tables



(a) TPC-H Lineitem (b) TPC-H Customer

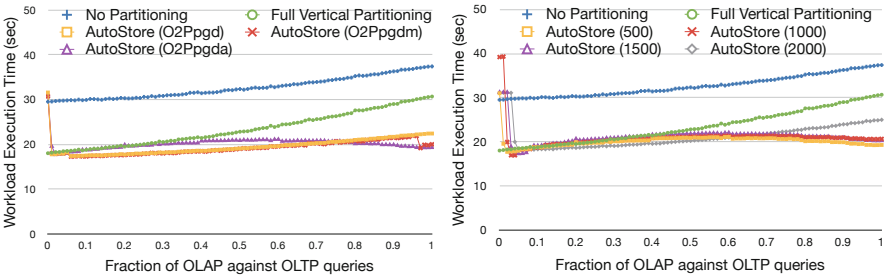
Fig. 2. Running times of different algorithms over changing workload type [100 queries each]



(a) TPC-H Lineitem (b) TPC-H Customer

Fig. 3. Running time of different algorithms over varying workload size [with 50% OLAP, 50%OLTP queries]

OLTP and OLAP-style queries. Figures 3(a) and 3(b) show the scalability of O^{2}P over TPC-H Lineitem and Customer tables respectively. We can see that all variants of O^{2}P algorithm scale linearly with the workload size. Hence, from now on we will only consider O^{2}Ppgd algorithm.



(a) Comparing different methods (b) Varying query window size

Fig. 4. Comparison of No Partitioning, Full Vertical Partitioning, and AutoStore in main-memory implementation

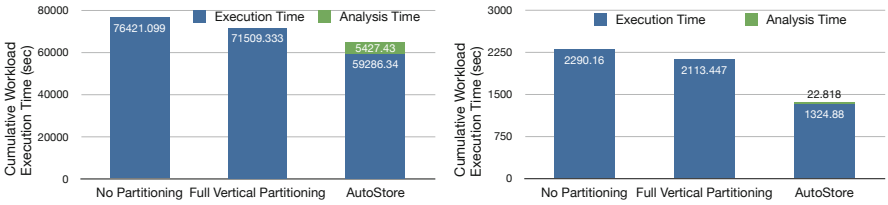
5.2 Evaluating Query Performance

Now we evaluate the query execution performance of AutoStore in comparison with No and Full Vertical Partitioning. In this evaluation we use a main-memory implementation of AutoStore in Java. In order to show how AutoStore adapts vertical partitioning to the query workload, we use a universal relation de-normalized from a variant of the TPC-H schema [28]. Similar as in [28], we choose a vertical partition with part key, revenue, order quantity, lineitem price, week of year, month, supplier nation, category, brand, year, and day of week for our experiments. Further, since we consider equal size attributes only, we map all attributes to integer values, while preserving the same domain cardinality. We use a scale factor (SF) of 1.

Figure 4(a) shows the performance of No Partitioning, Full Vertical Partitioning, AutoStore with O^{2}Ppgd, AutoStore with O^{2}P pgdm and AutoStore with O^{2}Ppgda. We vary the fraction of data accessed, i.e. both the attribute and tuple selectivity along the x-axis. We vary the OLTP/OLAP read access patterns as in Section 5.1, with a step size of 0.01%. From the figure we can see that AutoStore automatically adapts to the changing workload, i.e. even though it starts with no-partitioning configuration, AutoStore matches or improves full vertical partitioning performance. Therefore, from now on we consider only O^{2}Ppgda. Figure 4(b) shows the performance of AutoStore when varying query window size. From the figure we observe that larger query windows, e.g. query window of 2000 after 70% OLAP, become slower. This is because the partitioning analyzer has to now estimate the costs of more number of queries while analyzing partitioning schemes.

5.3 Evaluation over Real System

Modern database systems, e.g. PostgreSQL, have a very strong coupling between their query processors and data stores. This makes it almost impossible to replace the underlying data store without touching the entire software stack on top. This



(a) TPC-H Lineitem (b) TPC-H Customer

Fig. 5. Total running time of varying OLTP-OLAP workload on different tables in BerkeleyDB

limitation led us to consider BerkeleyDB (Java Edition), which is quite flexible in terms of physical data organization, for prototyping.

In BerkeleyDB we store a key-value pair for each tuple. The key is composed of the partition ID and the tuple ID, and the value contains all attribute values in that partition. Since BerkeleyDB sorts the data on keys by default, we simple need to change the partition ID to change the partitioning scheme. Again, we vary the OLTP/OLAP read access patterns as in Section 5.1, with a step size of 0.01% over TPC-H dataset (total size 1GB). For different layouts, Figure 5(a) shows the total query execution times over TPC-H Lineitem table and Figure 5(b) shows the total query execution times over TPC-H Customer table. In general, AutoStore outperforms the best layout in each of the tables. For instance, even though AutoStore starts from a no-partitioning configuration, it improves over Full Vertical Partitioning by 36% in Customer table.

6 Related Work

Offline Horizontal and Vertical Partitioning. Horizontal partitioning is typically done based on values (range, hash, or list). A recent work proposed workload-based horizontal partitioning [12]. However, it is still offline. Vertical partitioning started with early approaches of heuristic based partitioning [16] of data files. The state-of-the-art work in vertical partitioning [23] develops the notion of attributes affinity, quantifying attribute co-occurrence in a given set of transactions. This work creates a clustered attribute affinity matrix in the first step and applies binary partitioning repetitively in the second step. A follow-up work [24] presents graphical algorithms to improve the complexity of their approach. Other works took the type of scan into account to analyze the disk accesses [10] and formulated an integer linear programming problem to arrive at the optimal partitioning for relational databases [10]. Next, researchers proposed transaction based vertical partitioning [9], arguing that since transactions have more semantic meaning than attributes, it makes more sense to partition attributes according to a set of transactions. However, all of these works considered data partitioning as a one-time *offline* process, in contrast to the online approach to data partitioning in AutoStore. Recent works integrate partitioning into physical database design tuning problem [3] with the objective of

finding the configuration producing the minimum workload cost within the storage bound. This work first produces *interesting column groups* by applying a heuristic-based pruning of the set of all column groups. The column groups are then merged before all possible partitioning schemes are enumerated. These steps are not feasible in an online setting. HYRISE [15] analyzes the access patterns to partitions data. However, this approach is (1) still offline, i.e. it is not able to adapt to changing workloads, (2) restricted to main memory, whereas AutoStore works for both disk and main memory DBMSs, and (3) is limited to vertical partitioning, whereas AutoStore solves VPP and HPP equivalently.

Online Physical Tuning. Dynamic materialized views [31] materialize the frequently accessed rows dynamically. [6] proposes online physical tuning for indexes, without looking at the partitioning problem. Database Cracking [19] dynamically sorts the data in column stores based on incoming queries. However, these works still do not address the online partitioning problem.

7 Conclusion

In this paper, we revisited database partitioning with the objective of automatically fitting data to queries to an online query workload. We presented AutoStore, an online self-tuning database store. AutoStore monitors the workload and takes partitioning decisions automatically. We generalized VPP and HPP to the 1DPP. We presented the O^2P algorithm to effectively solve 1DPP. We performed an extensive evaluation of our algorithms over TPC-H data. We showed experimental results from a main-memory and a BerkeleyDB implementations of AutoStore over mixed workloads. Our results show that O^2P is faster than earlier approaches by more than two orders of magnitude, and still produces good quality partitioning results. Additionally, our results show that over changing workloads AutoStore outperforms existing stores.

Acknowledgements. Work partially supported by DFG, M2CI.

References

1. Agarwal, S., et al.: Database Tuning Advisor for Microsoft SQL Server 2005. In: VLDB (2004)
2. Agrawal, S., Chu, E., Narasayya, V.: Automatic Physical Design Tuning: Workload as a Sequence. In: SIGMOD (2006)
3. Agrawal, S., et al.: Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In: SIGMOD (2004)
4. Alagiannis, I., et al.: An Automated, Yet Interactive and Portable DB Designer. In: SIGMOD (2010)
5. Bruno, N., Chaudhuri, S.: Physical Design Refinement: The "Merge-Reduce" Approach. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 386–404. Springer, Heidelberg (2006)
6. Bruno, N., et al.: An Online Approach to Physical Design Tuning. In: ICDE (2007)

7. Bruno, N., Chaudhuri, S.: Constrained Physical Design Tuning. In: PVLDB (2008)
8. Chaudhuri, S., Narasayya, V.: An Efficient Cost-driven Index Selection Tool for Microsoft SQL Server. In: VLDB (1997)
9. Chu, W.W., Ieong, I.T.: A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems. IEEE TSE 19(8), 804–812 (1993)
10. Cornell, D.W., Yu, P.S.: A Vertical Partitioning Algorithm for Relational Databases. In: ICDE (1987)
11. Cornell, D.W., Yu, P.S.: An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. IEEE TSE 16(2), 248–258 (1990)
12. Curino, C., et al.: Schism: a Workload-Driven Approach to Database Replication and Partitioning. In: PVLDB (2010)
13. Dittrich, J., Jindal, A.: Towards a One Size Fits All Database Architecture. In: CIDR (2011)
14. Dittrich, J.-P., Fischer, P.M., Kossmann, D.: AGILE: Adaptive Indexing for Context-aware Information Filters. In: SIGMOD (2005)
15. Grund, M., et al.: HYRISE - A Main Memory Hybrid Storage Engine. In: PVLDB (2010)
16. Hammer, M., et al.: A Heuristic Approach to Attribute Partitioning. ACM TODS (1979)
17. Hankins, R.A., Patel, J.M.: Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In: VLDB (2003)
18. Hoffer, J.A., Severance, D.G.: The Use of Cluster Analysis in Physical Data Base Design. In: VLDB (1975)
19. Idreos, S., et al.: Database Cracking. In: CIDR (2007)
20. Jermaine, C., Omiecinski, E., Yee, W.G.: The partitioned exponential file for database storage management. The VLDB Journal 16, 417–437 (2007)
21. Jindal, A.: The Mimicking Octopus: Towards a one-size-fits-all Database Architecture. In: VLDB PhD Workshop (2010)
22. Kimura, H., et al.: CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. In: VLDB (2010)
23. Navathe, S., et al.: Vertical Partitioning Algorithms for Database Design. ACM TODS (1984)
24. Navathe, S., Ra, M.: Vertical Partitioning for Database Design: A Graphical Algorithm. In: SIGMOD (1989)
25. O'Neil, P.E., et al.: The Log-Structured Merge-Tree (LSM-Tree). Acta Inf. (1996)
26. Ozmen, O., Salem, K., Schindler, J., Daniel, S.: Workload-Aware Storage Layout for Database Systems. In: SIGMOD (2010)
27. Papadomanolakis, S., Ailamaki, A.: AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In: SSDBM (2004)
28. Raman, V., et al.: Constant-Time Query Processing. In: ICDE (2008)
29. Sacca, D., Wiederhold, G.: Database Partitioning in a Cluster of Processors. ACM TODS 10(1), 29–56 (1985)
30. Schnaitter, K., et al.: COLT: Continuous On-Line Database Tuning. In: SIGMOD (2006)
31. Zhou, J., et al.: Dynamic Materialized Views. In: ICDE (2007)
32. Zilio, D.C., et al.: DB2 Design Advisor: Integrated Automatic Physical Database Design. In: VLDB (2004)

Adaptive Processing of Multi-Criteria Decision Support Queries

Shweta Srivastava[1], Venkatesh Raghavan[2], and Elke A. Rundensteiner[3]

[1] Microsoft Corporation, One Memorial Drive, Cambridge, Massachusetts, USA
[2] Data Computation Division, EMC Greenplum, 1900 S Norfolk, San Mateo, CA, USA
[3] Department of Computer Science, Worcester Polytechnic Institute, Massachusetts, USA
shws@microsoft.com, venkatesh.raghavan@emc.com,
rundenst@cs.wpi.edu

Abstract. Business intelligence applications ranging from stock market tickers to strategic supply chain adaptation systems require the efficient support of multi-criteria decision support (MCDS) queries. Skyline queries are a popular class of MCDS queries that have received a lot of attention recently. However, a vast majority of skyline algorithms focus entirely on the input being a single data set. In this work, we instead focus on supporting the more powerful *SkyMapJoin* queries. Our Adaptive-SKIN framework conducts processing at two levels of abstraction thereby effectively minimizing the two primary costs, namely the cost of generating join results and the cost of dominance comparisons to compute the final skyline of the join results. Our proposed approach hinges on two key principles. First, in the input space – Adaptive-SKIN determines the abstraction levels dynamically at run time instead of assigning a static one at compile time. This is achieved by adaptively partitioning the input data driven by the feedback of the results already generated. Second, Adaptive-SKIN incrementally build the output space, containing the final skyline, without generating a single join result. Our approach selectively drills into regions in the output space that show promise in generating result tuples as well as avoiding the generation of intermediate results that do not contribute to the query result. In this effort, we propose a cost-vs.-benefit driven strategy for abstraction selection. Our experimental evaluation demonstrates the superiority of the Adaptive-SKIN over state-of-the-art techniques over benchmark data.

1 Introduction

Real-time applications, ranging from Internet aggregators, business intelligence to data warehouses require the support for complex multi-criteria decision support (MCDS) queries. Most decisions in business intelligence applications are multi-dimensional in nature that require analyzing the trade-offs between multiple factors to determine the most favorable set of options. Furthermore, in most applications decision support queries must be processed in a time-effective manner as delays can result in losing the competitive advantage.

Pareto-optimal (or *skyline*) queries are a popular class of multi-dimensional decision support queries [1]. Unlike traditional queries, whose goal is to return only exact matches, skyline queries return a set of non-dominated results meaning each result is

M. Castellanos, U. Dayal, and W. Lehner (Eds.): BIRTE 2011, LNBIP 126, pp. 81–101, 2012.
© Springer-Verlag Berlin Heidelberg 2012

better than the others in at least one criterion. Recently, we have witnessed a flurry of techniques [1–3] that evaluate skyline queries over a *single* homogeneous data set. The common assumption of viewing skyline as an operator on top of the traditional SPJ *select project join* queries applied on homogeneous data sets is rather limiting since a vast majority of MCDS applications in practice do not operate on just a single source. Instead, they require to (1) access data from disparate sources via joins, and (2) combine several attributes across these sources through possibly complex user-defined functions to characterize the final composite product. We motivate this work via the following real-world application scenarios.

Supply-Chain Management. A manufacturer in a supply chain aims to maximize profit, market share, etc., and minimize overhead, delays, etc. This is achieved by structuring an optimal production and distribution plan through the evaluation of various alternatives.

```
Q1: SELECT R.id, T.id, (R.uPrice + T.uShipCost) as tCost,
(2 * R.manTime + T.shipTime) as delay
FROM Suppliers R, Transporters T
WHERE R.country=T.country AND
P1' in R.suppliedParts AND R.manCap >=100000
PREFERRING LOWEST(tCost) AND LOWEST(delay)
```

$Q1$ identifies the suppliers that can produce "100K" units of the part "*P1*" and couples them with transporters that deliver it. The preference is to minimize both total cost ($tCost$) as well as delays ($delay$). In this work, we target such queries which combine the skyline and mapping functions over the join, here known as **SkyMapJoin** (SMJ) queries.

Internet Aggregators. The rapid increase in the number of on-line vendors has resulted in Internet aggregators such as Froogle[1], for durable goods, and Kayak[2] for travel services, are fast growing in popularity. Aggregators access and combine data form several sources to produce complex results that are then pruned by the skyline operation. Consider the query $Q2$, where the user is planning a holiday in Europe visiting both Rome and Paris. The user has different preferences in each leg of the journey. For instance since Rome is an ancient city, the user is willing to walk twice as much in Rome than in Paris. In addition, the user has a cumulative goal of minimizing the total cost of the trip.

```
Q2: SELECT R.id Rome,T.id Paris, (R.price + T.price) as cost,
(2 * R.distance + T.distance) as distance
FROM RomeHotels R, ParisHotels T
PREFERRING LOWEST(tCost) AND LOWEST(tDistance)
```

Beyond the above mentioned business intelligence applications, there are numerous other applications in the field ranging from *drug discovery* to *query relaxation* that

[1] http://froogle.google.com/shoppinglist
[2] www.kayak.com

require similar MCDS queries. In this work we target queries involving both skyline and mapping operations over disparate data sources, known as *SkyMapJoin* queries introduced by [4,5].

State-of-the-Art Techniques. Existing techniques view the skyline operation as an add-on after the join and thus follow a *join-first skyline-later* (JF-SL) paradigm [1]. JFSL divides query evaluation into two steps, namely first produce all possible join results, and second perform skyline evaluation over the entire join results. This approach misses several optimization opportunities. JFSL spends precious resources in producing join results that may not belong to the final skyline result set. Furthermore it also bears the cost of comparing all the unwanted join objects. In [4] we introduced a method called *SKIN* that leverages this opportunity by considering both skyline dominance and mapping function transformation knowledge as part of the join processing logic. SKIN has been shown to outperform existing methods in some cases as much as several orders of magnitude. Similar to other partition-based techniques [6,7] the effectiveness of SKIN rests on the assumption that the level of partition abstraction is given by empirical results. We will show that a bad choice in the abstraction level can hinder the performance of an otherwise efficient framework.

Our Proposed Solution. In this work, we propose **Adaptive-SKIN** that successfully leverages the strengths of SKIN while at the same time alleviating its weaknesses by introducing an automated, dynamic decision making process. *Adaptive-SKIN* works on two data spaces namely, the *input space* containing the input tuples and an *output space* containing the intermediate join results and the final skyline results. In *Adaptive-SKIN* rather than processing the input and output spaces at a single rigid level of abstraction, we dynamically build the spaces by adaptively populating it with heterogeneous sized abstractions.

The contributions of this work can be summarized as follows: (1) We propose *Adaptive-SKIN* framework that provides an adaptive execution strategy to evaluate *SkyMapJoin* queries. (2) We present a *cost-vs-benefit driven* reasoning strategy that aids Adaptive-SKIN in exploring the input space based on the feedback from the abstract output space. (3) Our experimental evaluation demonstrates that Adaptive-SKIN shows superior performance over state-of-the-art techniques.

2 Background: The SKIN Approach

We briefly review the background need for this work. In particular, Adaptive-SKIN is built on top of SKIN [4] which we review here. The fundamental approach in SKIN is to solve the skyline problem at a coarse level of abstraction before solving it at the expensive level of individual tuples. The key idea is to have two layers of abstraction, namely *macro-level* and *micro-level processing*. Table 1 summarizes the notation used in this work.

Macro Level Processing. The aim of this step is to perform query execution at a higher granularity. For this, we generate the abstract output space. For this, SKIN assumes each input relation is uniformly partitioned into an equi-sized grids. For a pair of input cells,

Table 1. Notations Used In This Work

Notation	Meaning
$r_i \succ_P r_j$	Tuple r_i **dominate** tuple r_j in all attribute dimensions $a_k \in P$
$r_i \succ r_j$	Tuple r_i **dominate** tuple r_j for all attribute dimensions
I_i^R	Input cells in R
$\mathcal{I}^R$	Set of all input cells in R
$r_f t_g$	Join result, $r_f \in R$; $t_g \in T$
$\mathcal{R}_{i,j}$	Region of output space that map the join results from the input cells $[I_i^R, I_j^T]$
$\mathfrak{R}$	Set of all regions in the output space
LOWER(X)	*Lower-bound* point of a region or cell
MARK(O_i)	Mark cell O_i as "*non-contributing*"
IS_MARKED(O_i)	*True*, if cell O_i is marked, else *false*.

one from each table $I_a^R \in R$ and $I_b^T \in T$, we: (1) investigate whether the tuples in these cells produce at least one join result, and if so (2) determine the *region* of the output space into which the generated join results will fall (denoted as $\mathcal{R}_{a,b}$). To illustrate, let us consider a scenario with the domain values of the join attributes are finite and known. In such a scenario, for each input cell we maintain a list of domain values of the join attribute(s) for the tuples mapped into that particular partition. Therefore, if two cells share at least one join domain value we can guarantee that their join will result in at least one join result. The full treatment of joins is described in [4]. Henceforth we only need to consider these output regions which are guaranteed to be populated for further processing.

Example 1. Tuples in the input cell from Supplier (R), I_1^R with bounds $[(0,4)(1,5)]$, when joined with tuples in input cell $I_2^T[(0,4)(1,5)]$ from Transporter (T), will result in join results that will fall in the region bounded by the lower-bound point $b(3,5)$ and the upper-bound point $B(6,7)$ in Figure 1. This output region is denoted as $\mathcal{R}_{1,2}$.

In query $Q1$ the preference is to minimize all skyline-dimensions. In a pessimistic scenario for each output region $\mathcal{R}_{i,j}$, all the intermediate join tuples and then mapped results would lie on the upper-bound point of $\mathcal{R}_{i,j}$. $SKIN$ introduces the notion of the **pessimistic output skyline**, denoted as $\mathcal{S}_{pes}$ to identify all output regions that will definitely be dominated and therefore can be removed before even processing its tuples. For any region in $\mathcal{R}_{i,j} \in \mathfrak{R}$ if $\exists s \in S_{pes}$ such that $s \succ$ LOWER($\mathcal{R}_{i,j}$) then no intermediate result $r_f t_g \in \mathcal{R}_{i,j}$ can be contained in the output skyline. Thus, the pessimistic skyline reduces the comparisons of output regions generated to only those regions that belong to S_{pes}.

Micro Level Processing. Having effectively eliminated higher abstractions, this step executes tuple-level joins and tuple-level dominance comparisons to produce the resulting skyline.

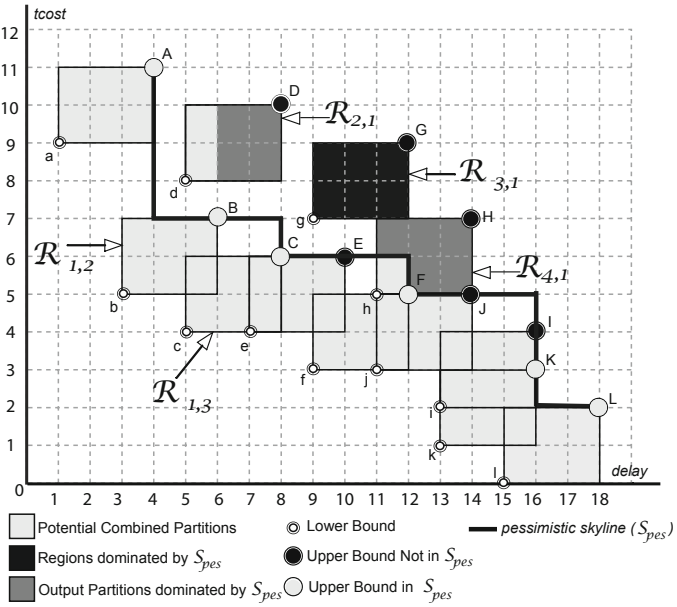


Fig. 1. Macro-Level Processing

3 Motivation

In this section, we explore the different parameters that affect the execution time of SKIN.

3.1 Costs of Macro Level Processing

This phase incurs the cost of comparing every region that is generated with all regions that lie in the pessimistic skyline across all skyline dimensions. This is found to be $O(N_{total} \cdot |S_{pes}| \cdot d)$, where N_{total} is the total number of regions generated, $|S_{pes}|$ is the number of regions in the pessimistic skyline and d is the number of attributes in the skyline result as stated by the query. For the sake of simplicity we consider that d attributes from each relation combine to form d skyline attributes as in Q1. If k_i is the number of partitions that each of the dimensions in the input is divided into, then the total number of cells in each relation is k_i^d. Therefore, $N_{total} = \sigma k_i^{2d}$ where σ is the join factor. Next, the cost of computing the pessimistic skyline is $O(N_{total}^2)$. Thus, the total cost of performing *macro level processing* corresponds to the cost of creating the regions and maintaining the pessimistic skyline $|S_{pes}|$.

$$O\left(k_i^d \cdot k_i^d\right) + O\left(\sigma^2 \cdot k_i^{4d}\right) \tag{1}$$

3.2 Costs of Micro Level Processing

This phase incurs the costs of processing individual tuples in each output region which includes creating combined objects from the two input cells I_a^R and I_b^T that form region $R_{a,b}$ and subsequently performing skyline comparisons on these objects to produce the final skyline result set. For a single region $\mathcal{R}_{a,b}$ the cost of *micro-level-processing* is:

$$\begin{aligned} Cost(\mathcal{R}_{a,b}) &= C_{join}(\mathcal{R}_{a,b}) + C_{map}(\mathcal{R}_{a,b}) + C_{sky}(\mathcal{R}_{a,b}) \\ &= O(\sigma \cdot n^2) + O(\sigma \cdot n^2 \cdot d) + O(\sigma \cdot n^4 \cdot d) \\ &= O(\sigma \cdot n^4 \cdot d) \end{aligned} \tag{2}$$

where $n = |I_a^R| = |I_b^T|$.

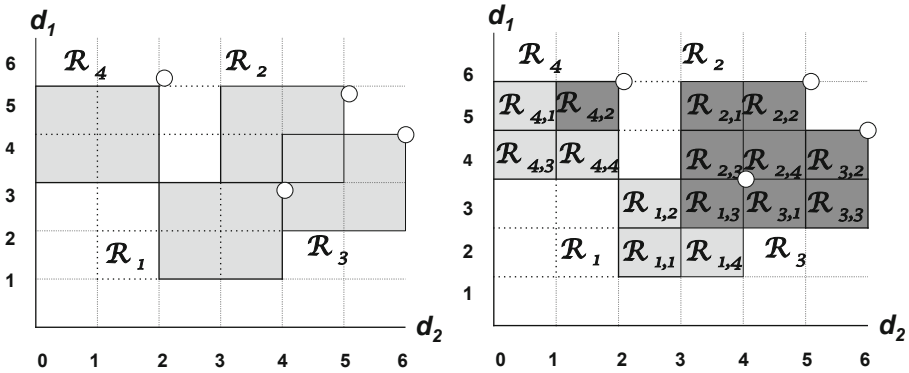
Therefore the cost of materializing all N_{remain} regions is $N_{remain} \cdot Cost(\mathcal{R}_{a,b})$. Let N_{remain} be the number of regions that remain to be materialized after pruning all dominated regions during *macro level processing*.

3.3 Factors Affecting Macro- and Micro-Level Processing Costs

Following are some of the important factors that affect adversely the execution costs of SKIN. First, the total number of regions created during marco-level processing N_{total} depends on k_i (the number of partitions that each of the dimensions in the input is divided into). The later is in turn determined by the partition size δ that the input is uniformly divided into. Second, the size of the pessimistic skyline $|S_{pes}|$ which is sensitive to the data distribution. Third, the cardinality n of each input cell.

As described above in Equations 1 and 2, *macro level processing* is quadratic in the number of cells k_i while *micro level processing* is quadratic in the number of tuples in each cell n. Therefore to reduce the total cost the goal should be to minimize both k_i and n. However, these two values are inversely proportionate to one another. That is, for an independent distribution, a smaller number of tuples in a cell means that each cell covers a smaller area and therefore the number of cells are higher than in a scenario where the number of tuples are high. From this analysis it is evident that if we try to decrease the costs of *micro level processing* the costs of performing *macro level processing* increases and vice versa. Since a reduction of both cannot be achieved at the same time, one of the goals in this work is to strike the right balance between n and k_i. We will now show with the help of examples how selecting an appropriate δ can affect the cost of skyline computation.

Case I - SKIN with Larger δ (Static Approach). Figure 2.a denotes the output space after *macro-level processing* where $N_{total} = N_{remain}$. In other words the regions have a fairly large size such that no region can be eliminated during *macro-level processing*. For this example, SKIN will have to bear an additional cost to perform *macro level processing* without seeing any of its benefits. In addition, since the regions are at a higher granularity, the cost of *micro level processing* is going to be high as the number of tuples in its cells n will be high (assuming independent distribution). The challenge is to find out whether changes to the granularity of the cells will likely reduce the overall costs.



(a) Output space with pre-determined δ

(b) Output space with smaller δ

(c) Output space with heterogeneous δ

Fig. 2. Effect of varying δs on the output space

Case II - SKIN Smaller δ (Static Approach). Figure 2.b shows the same output space with smaller sized regions. This implies that the δ used to partition datasets R and T is smaller as compared to that used in Case I. In this scenario, several output regions are completely dominated and therefore can be safely eliminated. Although a larger number of regions go into *micro level processing* as compared to Case I, their respective cardinality is smaller. This implies that the costs of processing each one of them have been drastically reduced. However, this reduction of costs has not come without spending extra resources. *Macro level processing* has to do more work to generate an increased number of regions and to perform more skyline comparisons to maintain the pessimistic skyline $\mathcal{S}_{pes}$ as compared to Case I. It is also evident that attaining the right granularity of output regions is an important factor in achieving effective elimination and with it the side-effect of reduced costs. In other words, it appears that if we could determine the most appropriate level of abstraction a-priori it would lead to an ideal scenario. However the selection of a correct δ for partitioning the input side without any knowledge of the output remains a challenging problem and therefore SKIN assumes that a δ is given a priori.

Case III - Adaptive-SKIN (Heterogeneous δ) Approach. In the previous example the additional costs of maintaining $\mathcal{S}_{pes}$ may not be worth bearing if it does not translate into more benefit. Furthermore for certain data distributions, lower granularity of input cells does not necessarily increase the pruning capacity of all output regions in the same proportion. In other words for some areas in the output space making smaller regions may not reap any benefits of elimination. In Figure 2.c that represents Case III the first visual difference that comes is that unlike Cases I and II, where all regions were of the same size, here the regions are of variable size. Output regions of heterogeneous sizes are a result of a heterogeneously partitioned input space. In Case III, while the area eliminated is the same as in Case II, the cost of *macro level processing* is lower since the number of total output regions created and compared for elimination are fewer. Since the cost of *macro level processing* is affected quadratically by the number of input cells, this is a considerable cost savings.

The two key observations of the above analysis are as follows. First, finer granularity output regions increase elimination potential, thereby reducing the subsequent *micro level processing* costs. However the associated overhead can be cost prohibitive. Thus it is important to find the right granularity of regions such that they are small enough to be able to eliminate more regions and large enough to keep its macro-level processing costs low. Second, since the static approach has no feedback mechanism it keeps making incorrect choices in partitioning. A feedback driven approach is crucial in increasing the pruning capacity while limiting additional processing costs.

4 Our Proposed Adaptive-SKIN Framework

We now present our *adaptive evaluation framework* called **Adaptive-SKIN** as illustrated in Figure 3. Our proposed framework builds on top of SKIN making SKIN δ-insensitive by empowering its *macro-level processing* to arrive at the right granularity, or in other words heterogeneous δ, depending on the result distribution in the output

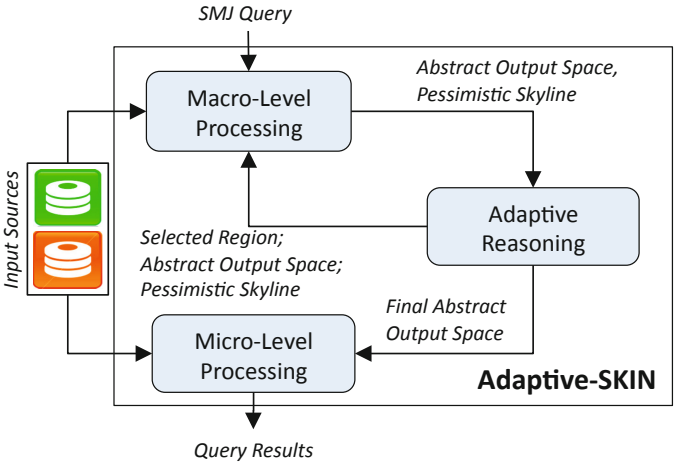
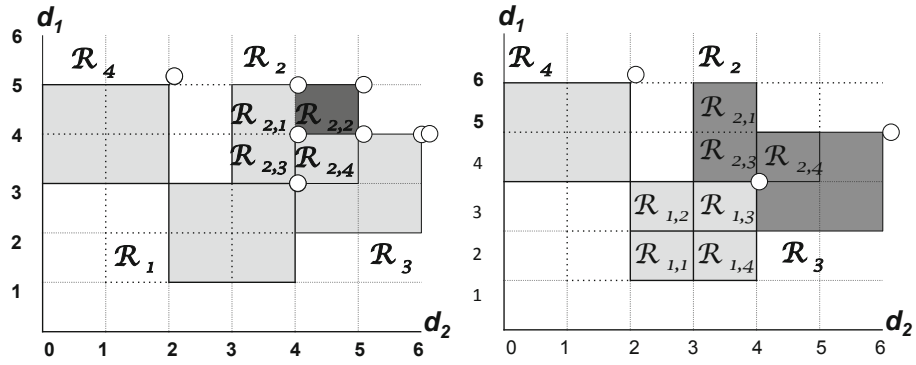


Fig. 3. Adaptive Evaluation Framework

space. The *macro-level processing* takes as input the datasets (R and T), the preference query and a *coarse* δ (for instance $\delta = 50$). Next, the *adaptive reasoning* module looks ahead into the coarse grained output space to iteratively pick the next output region to zoom in. The selected region when made finer will result in the maximum pruning among all its peers. Armed with this knowledge, the *macro-level processing* creates smaller output regions by repartitioning its associated input partitions and updating the output space. This iterative process continues till the abstract processing cannot yield more pruning and the overhead costs outweighs its benefits from elimination. At this point, the output space is shipped for *micro level processing*. In a nutshell, we adapt selected areas of the output space over several iterations to obtain maximum possible pruning in a coarse abstraction of the output space to minimize the cost of *micro level processing*.

In the following discussion, we address two key questions that our approach poses: (1) *which* output region to zoom into. and (2) *when* to stop abstract-level processing and initiate *micro-level processing*.



(a) Region R_2 selected for repartitioning (b) Region R_1 selected for repartitioning

Fig. 4. Effects of Region Selection

Region Selector. In each iteration, this component selects the next most-beneficial region to adapt. Figure 4.a motivates the importance of selecting regions in an appropriate order. Consider the following example where we employed a random selection process and pick region R_2. After repartitioning we added to the space $R_{2,1}$, $R_{2,2}$, $R_{2,3}$ and $R_{2,4}$. Since $R_{2,2}$ is the only region completely dominated by another region ($R_{2,3}$), it is eliminated. In contrast, let us assume we choose R_1 instead. As it is evident in Figure 4.b that on repartitioning R_1 more of the output space can be pruned. In next section, we present a cost-vs-benefit driven *dominance potential* metric used in selecting the next region for further processing.

Threshold Analyzer addresses the second problem of keeping regions small enough to facilitate more elimination but large enough to keep the cost of repartitioning low. The *threshold analyzer* uses the cardinality of the participating input partitions of a region as a good metric to stop repeated macro-level processing. The reasoning is that a small but dense output region has more potential for pruning therefore it is deemed

worthwhile for further investigation. Once all output regions reach the threshold limit the *micro level processing* takes over.

Output Region Generator is part of *macro-level processing* that updates the output space after every feedback provided by the abstract reasoning. It first combines input partitions based on their join attributes by applying *mapping functions*. Next, it maintains the pessimistic skyline and eliminates regions based on dominance comparisons.

Adaptor component does the actual work of creating smaller output regions. This is achieved by re-partitioning the input partitions of the selected region and sends the newly created input partitions to the output region generator. This process continues in an iterative manner until all regions have not reached the threshold size. The reason why we adapt the input space is because if we simply divide output regions without considering the input space, there is no way of knowing whether a dominating region will not be empty when we start *micro-level processing*. In case a dominating region is empty, it might end up eliminating a region that could have potential skyline candidates – leading to an incorrect result set.

4.1 The Overall Process

Figure 5 illustrates the control and data flow in our proposed Adaptive-SKIN framework. As part of *macro-level processing* the *output region generator* creates an abstract output space starting with a coarse δ. We store each output region $\mathcal{R}_{a,b}$ that is

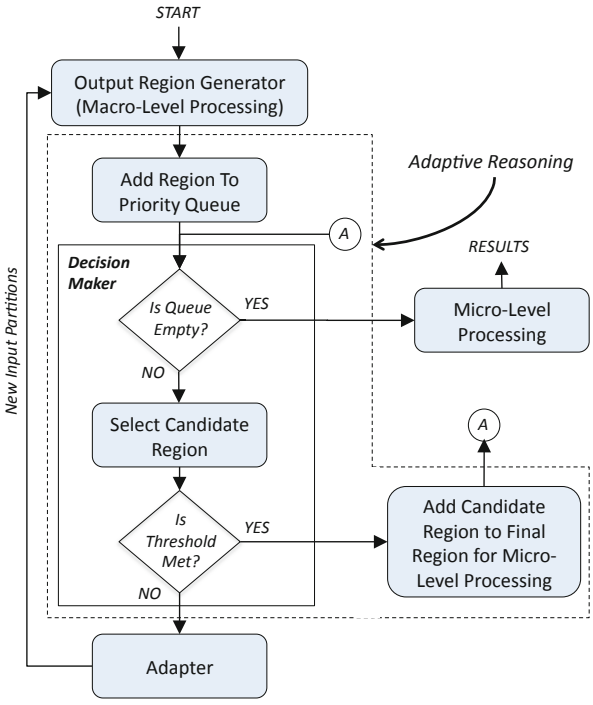


Fig. 5. Control Flow in Adaptive-SKIN

generated is maintained in a central priority queue (P). The queue is sorted by the pruning benefits of each output region $\mathcal{R}_{a,b}$. The *region selector* sends the candidate output region at the top of the queue to the *threshold analyzer*. The job of the analyzer is to verify whether or not the further repartitioning of the candidate region is estimated to be beneficial. If further repartitioning is indeed beneficial, the chosen region is sent to the Adaptor to repartition its input cells. It is then passed to the *output region generator* for generating newer output regions generated from combining the repartitioned input cells. This process is repeated till no regions need to investigated. In the scenario where the candidate region has crossed the threshold, it is removed from the queue. This indicates that the overhead cost of adapting the region is deemed larger than its benefits. The candidate region is now transferred to a list called R_{final} that is eventually passed on to micro-level processing. Now that the region has been removed other regions will get an opportunity to be investigated. This process continues till the queue becomes empty. This indicates that all regions have crossed the threshold and will no longer provide more benefits from pruning than the overhead cost of repartitioning. At this point R_{final} is passed on to micro-level processing for computing the final skyline result set.

4.2 Output Region Generator

This module accomplishes the operations of eliminating the dominated output regions with the help of the pessimistic skyline, S_{pes}. It incrementally maintains the pessimistic skyline and at the same time eliminates regions that are dominated by S_{pes}. Any region that is not part of S_{pes} but remains un-dominated is stored in a temporary output region list called R_{temp}. The first pass generates output regions and performs incremental S_{pes} maintenance. By incremental maintenance we mean the following. Each newly generated region is only compared with regions in S_{pes} to check whether it is going to be eliminated. If it is not, then there is a possibility that it may be added to S_{pes}. Furthermore, it is also possible that it displaces an output region from S_{pes} to R_{temp}. At the end of this first pass we get the final S_{pes} as all newly generated regions have been compared against it. At this point there may be some output regions in R_{temp} that are by now completely dominated but are still not eliminated because they were never compared to the incoming regions. The second pass which we call the *one pass scan* identifies these dominated regions and eliminates them by performing dominance comparisons between all regions in R_{temp} with S_{pes}.

4.3 Decision Maker

Region Selector. Before diving into the details of how the Region Selector works, let us understand how it impacts the cost of the entire adaptive process. In order to repartition a region, its input cells are repartitioned using a smaller δ. The cost of this repartitioning can be given as $O((n_a^R \cdot d) + (n_b^T \cdot d))$ where n_a^R and n_b^T are the number of tuples in the input cells I_a^R and I_b^T respectively (Refer Table 1 for notations). Thereafter the cost of combining input cells to create new output regions is $O(\sigma k_{new}^{2d} \cdot d) = O(N_{new} \cdot d)$ where N_{new} is the number of new output regions created after repartitioning, k_{new} is

the number of new cells that each dimension of existing partition is divided into using the new partition size δ_{new}.

Having created the regions they now need to be compared for elimination. The costs of comparing N_{new} with the pessimistic skyline S_{pes} for region-level elimination is $O(N_{new} \cdot |S_{pes}| \cdot d)$. If *all* output regions of a given iteration, lets call it $N_{fPrevious}$ were to be re-partitioned, the total cost of generating an entirely new output space with finer granularity would be $O(N_{fPrevious} \cdot ((|I_a^R| \cdot d) + (|I_b^T| \cdot d)) + (N_{new} \cdot d) + (N_{new} \cdot |S_{pes}| \cdot d)+))$. By the above analysis it is evident that fewer the number of regions repartitioned, the smaller the overhead costs. The *region selector* selects the output regions that would give maximum benefits of elimination. In doing so, it uses certain characteristics or metrics associated with the region that provides knowledge of its pruning capacity.

This technique is a cost saver for the following two reasons. One, by repartitioning only selected regions in iteration $i+1$, all resources spent on generating the output space in the previous iteration i are not lost. Two, in any given iteration no resources need be spent on regions that will result in zero or limited number of regions being eliminated.

We now discuss how the *region selector* goes about selecting the right regions and provide cost savings. At the end of the *one pass scan* all regions from R_{temp} and S_{pes} are added to the priority queue P that is part of the *region selector* component. These are the regions that have remained un-dominated and will now be considered for repartitioning. In every iteration one region will be selected for repartitioning. In order to make this selection we rank all regions based on their **dominance potential** metric. *Dominance potential* of a region $\mathcal{R}_{a,b}$ is defined as the number of regions that are dependent on it being processed. The higher the number of dependents higher is the region's dominance potential. A region $\mathcal{R}_{a,b}$ is said to be *dependent* on region $\mathcal{R}_{e,f}$ if LOWER($\mathcal{R}_{a,b}$) $\succ$ LOWER($\mathcal{R}_{e,f}$). When the output space is computed for the first time each region is compared with every other region to create a list dependents regions and the list of regions it depends on. Based on count of the dependents list, each region $\mathcal{R}_{a,b}$ is stored in the priority queue P for selection. The cost incurred in the first iteration is $O(N_f^2 \cdot d)$ where N_f is the number of regions after all possible elimination in the first iteration. Thereafter, in subsequent iterations, the *one pass scan* of the *output region generator* is replaced by a *Dependent Region Elimination Scan (DRES)* – which instead of comparing against all output regions only compares the dependents of the region with the newly created regions. Whenever a dependent region is eliminated, we create a ripple effect such that the count of all dependents and depend-on of this eliminated region is reduced by one. The cost of performing this maintenance operation for a selected region is $N_{DRES}(O(N_{dependent} + N_{depend-on}))$ where N_{DRES} is the number of dependents of the region and $N_{dependent}$ and $N_{depend-on}$ are the number of dependents and depend-ons of the eliminated region. In the process of comparing new regions to their parent region's dependents their own dependent list is generated. The cost of computing dependents of these newly created regions is $O(N_{DRES} \cdot N_{new} \cdot d)$ where N_{new} is the number of new regions that remain un-dominated by S_{pes}.

Threshold Analyzer. Once a region is selected, the *threshold analyzer* component verifies it against the threshold value. In this work, we define *threshold* in terms of the size of the input cells of a region. If the number of tuples in the input partitions is smaller

than the *threshold* the output region is removed from the priority queue. It is then passed on to the *adaptor* for repartitioning its input partitions.

4.4 Adaptor

This is responsible for repartitioning the newly selected regions. Consider that during one of the iterations, the region $\mathcal{R}_{a,b}$ created is selected. To achieve this, we repartition the regions' participating input cell ($\mathcal{I}_a^R \in R$ and $\mathcal{I}_b^T \in T$) by first determining a smaller δ. For the sake of simplicity in this work we have chosen the new δ also called δ' to be half of the previous δ, i.e. $\delta' = \delta/2$. Clearly any other more complex metric for selecting δ could easily be plugged into our proposed Adaptive-SKIN framework. After repartitioning the new input cells are sent to the *output region generator* to create new output regions in place of its parent. Since $\mathcal{R}_{a,b}$ is now adapted, we delete it from memory.

However the original input cells $\mathcal{I}_a^R \in R$ and $\mathcal{I}_b^T \in T$ contributing to region $\mathcal{R}_{a,b}$ cannot be deleted for the following reasons. The input cell in relation $\mathcal{I}_a^R \in R$ combines with all input cells of relation T to create a set of output regions. Therefore it is possible that even after elimination of the region $\mathcal{R}_{a,b}$, the cell $\mathcal{I}_a^R \in R$ may still belong to more than one output region. Also, if subsequently the region $R_{a,c}$ is selected for repartitioning it need not re-create the child cells of I_a^R, but can directly access its children created earlier. We therefore store the child cells in a hierarchical structure for future use.

5 Discussion

In this work, we target multi-dimensional applications (such as those outlined in the introduction) where the number of skyline dimensions is in the order of 2 to 6 dimensions. The reasoning here is that as the number of attributes on which the skyline is applied to increases the cardinality of the output results also increases drastically. Human analyst cannot make effective decisions when faced with such high cardinality result sets. Thus, further processing such as linear ranking of results or clustering would need to be employed [8]. Increasing the usability of the results is an orthogonal problem to the one addressed in this work. High-dimensional skyline evaluation (even for single sets) has exponential time complexity [1]. Therefore, to extend this effort to high-dimensional applications with say 100s of dimensions is clearly a challenging task. However, the focus of this work is low-dimensional applications, we leave high-dimensional applications for our future work.

6 Performance Study

We verify the effectiveness of our proposed *Adaptive-SKIN* approach by comparing against state-of-the-art $JFSL^+$ [9], *Skyline Sort Merge Join* (*SSMJ*) [10] and *SKIN* [4].

Experimental Platform. All experiments are conducted on a Linux machine with AMD 2.6GHz Dual Core CPUs and 4GB memory. All algorithms are implemented in Java 1.5.0_16. In our analysis we use the total execution time as the comparison metric.

Benchmark Data. We conduct our experiments using data sets generated by [1]– *de-facto* standard for stress testing skyline algorithms. The data sets contain three extreme

attribute correlations, namely *independent*, *correlated*, or *anti-correlated*. For each data set (R and T), we vary the number of skyline dimensions d [2-6] while keeping the cardinality N constant at [500K]. The attribute values are real numbers in the range of [1–100]. The join selectivity σ is varied in the range $[10^{-3}$–$10^{-1}]$. We set $|R| = |T| = N$.

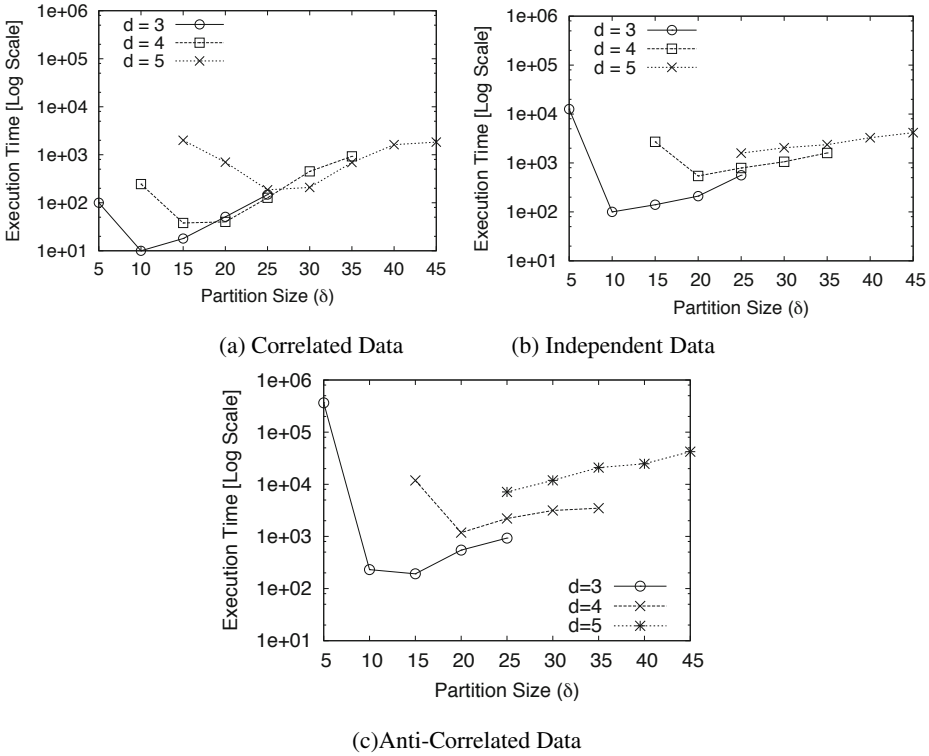
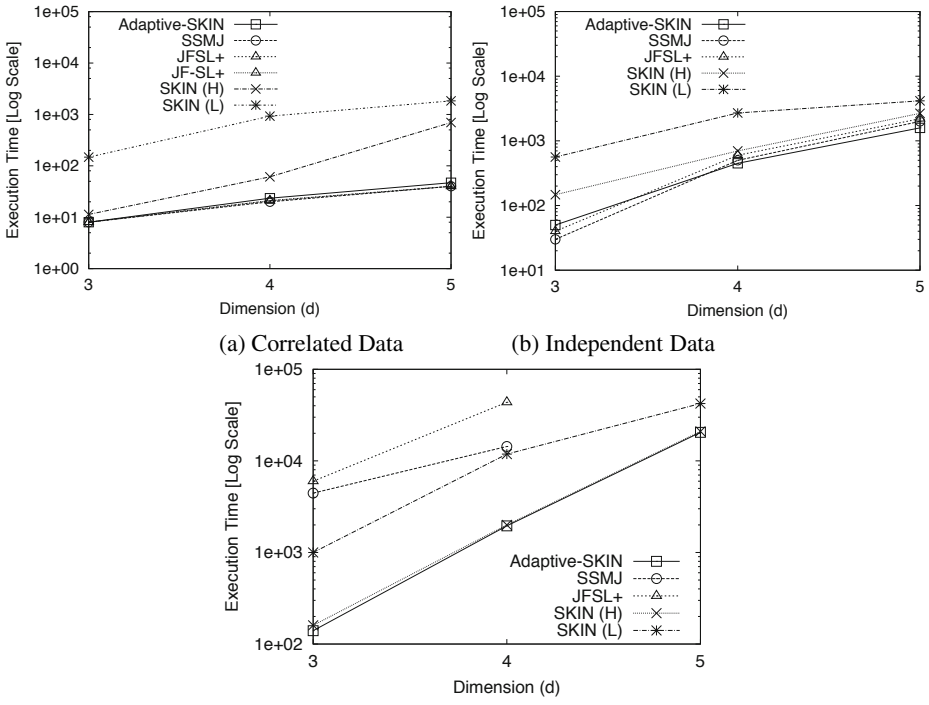


Fig. 6. Effects of δ on SKIN ($N = 500K$ $\sigma = 0.01$)

Experimental Analysis of SKIN. We first experimentally show the effects of varying partition sizes δ for the SKIN approach. As illustrated earlier in Section 3 the effectiveness of SKIN greatly depends on the ratio between the tuple-level vs. the abstract-level granularity. Figures 6.a, 6.b and 6.c show the execution time of SKIN for correlated, independent and anti-correlated data sets respectively. Smaller partition sizes result in many sparsely populated input partitions, and therefore the overhead costs of macro-level processing will out-weigh its benefits. This is evident for all three distributions (see Figure 6.a-6.c). Alternatively, as δ is increased the execution costs of macro-level processing is reduced, reflected by the dip in execution time. Larger δ however may only marginally reduce the number of combined objects generated but will increase the number of combined objects to be compared against the output space. This is depicted by the slow rise in execution costs as δ increases. Results in Figures 6.a-6.c imply that SKIN is sensitive to δ.



(a) Correlated Data (b) Independent Data

(c)Anti-Correlated Data

Fig. 7. Execution Time Comparison: Adaptive-SKIN vs. state-of-the-art techniques JFSL+ [9], SSMJ [10], and SKIN [4]; ($N = 500K$ $\sigma = 0.01$)

Adaptive-SKIN vs. SKIN. Next, we compare SKIN against our proposed Adaptive-SKIN approach. In the case of SKIN we have measured its execution time with two δ. SKIN (H) is measured with a high performing δ setting while SKIN (L) is measured with a low performing sub-optimal δ. Figures 7.a-7.c compare the execution times of Adaptive-SKIN vs. SKIN for varying dimensions $d[3-5]$ and data distributions.

For correlated data, a few tuples can dominate large portions of the intermediate results generated. In such a scenario have optimal δ ensures that the amount of elimination is maximized while costs are minimized. Figure 7.a reflects that Adaptive-SKIN by performing heterogeneous partitioning of both the input and output space can reach up-to 1 order of magnitude faster that SKIN (H). For independent distribution and $d = 5$, the adaptive approach is about 20% faster than SKIN (L) and marginally better than SKIN (H). Anti-correlated data is particularly hard on skyline algorithm. This is highlighted by the fact that both SSMJ and JFSL+ easily run out of memory space for $d = 5$. Adaptive-SKIN and SKIN(H) perform 100% better than the scenario when a poor δ is chosen, i.e., SKIN (L).

Experimental Conclusions. The main findings of our performance study can be summarized as follows: (1) Adaptive-SKIN is robust to all three distributions. (2) Adaptive-SKIN outperforms the low performing SKIN (L) for all data distributions. (3)

For correlated data sets, Adaptive-SKIN even quickly reaches the optimal abstraction-level to outperform SKIN (H). (4) Our proposed approach builts on top SKIN making it independent of a pre-determined δ selection. Our results confirm that Adpative-SKIN is superior or equal to SKIN in all setting and thus is solution of choice.

7 Related Work

Skyline Algorithms over Disparate Sources [1] presented a brief discussion of handling *"multi-relational skyline operator"* – skyline over joins. In the context of returning meaningful results by relaxing user queries, [9] presented various strategies that follow the *"blocking"* paradigm known here as join-first, skyline-later (JF-SL). Recently [5,10] have begun addressing the problem of efficiently handling skylines over joins. [10, 11] proposed *SSMJ* (*Skyline Sort Merge Join*) technique to handle skylines over join by primarily exploiting the principle of *skyline partial push-through*. This approach suffers from the following three drawbacks. First, *SSMJ* is only benefitial when the local level pruning decisions can successfully prune a large number of objects, like skyline friendly data sets such as correlated and independent data sets or very high selectivity [12]. Second, since they do not have any knowledge of the mapped output space, similar to *JF-SL*, *SSMJ* is unable to exploit this knowledge to reduce the number of dominance comparisons. Third, the guarantee that objects in the set-level skyline of an individual table clearly contribute to be in the output no longer holds here. This is so because they do not consider mapping functions which can affect dominance characteristics. Following the principles proposed in [10], [13,14] recently proposed alternative sorting based techniques.

Adaptive Techniques in Skylines [15] proposes an adaptive algorithm for controlling the degree of parallelism and the required network traffic while computing skyline results in a distributed environment. [16] too addresses the skyline computation for single data sets in a distributed environment but for higher dimensional data sets. They use dimension reduction techniques to reduce the dimensions to 1 single dimension. These techniques however limit themselves to processing skylines over a single data set.

Top-K or Ranked Queries retrieve the best K objects that minimize a user-defined *scoring function* to name a few [17, 18]. That is, from a *totally ordered* set of objects such queries fetch the top K objects, where the ordering criterion is a *single* scoring function. In contrast, the skyline operator returns a set of non-dominated objects based on *multiple* criteria in a multi-dimensional space and from a *strict partially ordered* set of objects. Therefore, the objects returned by Top-K may not be part of the skyline [2], or vice versa.

8 Conclusion

The efficient evaluation of skyline over disparate sources is burdened by two primary component cost factors, namely the cost of generating the intermediate join results and the cost of dominance comparisons to compute the final skyline of join results. State-of-the-art techniques handle this by primarily relying on making local pruning decisions

at each source. Although SKIN overcomes these drawbacks, it suffers from the dependence of its performance on a pre-determined abstraction level. This may lead to poor performance if the chosen level is poor. We propose Adaptive-SKIN which dynamically chooses the abstraction level of output spaces based on knowledge of their pruning capacity. This helps in attaining a balance between the benefits of effective elimination in the output space and the cost of adapting the abstraction levels.

Acknowledgment. This work is supported by the National Science Foundation under Grant No. IIS-0633930 and CRI-0551584. We thank Dr. Donald Kossmann for the synthetic data generator, which is the de-facto benchmark data sets for skyline evaluation.

References

1. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE, pp. 421–430 (2001)
2. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: SIGMOD Conference, pp. 467–478 (2003)
3. Bartolini, I., Ciaccia, P., Patella, M.: Salsa: computing the skyline without scanning the whole sky. In: CIKM, pp. 405–414 (2006)
4. Raghavan, V., Rundensteiner, E.A., Srivastava, S.: Skyline and mapping aware join query evaluation. Inf. Syst. 36(6), 917–936 (2011)
5. Raghavan, V., Rundensteiner, E.A.: Progressive result generation for multi-criteria decision support queries. In: ICDE, pp. 733–744 (2010)
6. Morse, M., Patel, J., Grosky, W.: Efficient continuous skyline computation. Inf. Sci., 3411–3437 (2007)
7. Mishra, C., Koudas, N.: Interactive query refinement. In: EDBT, pp. 862–873 (2009)
8. Chan, C.-Y., Jagadish, H.V., Tan, K.-L., Tung, A.K.H., Zhang, Z.: On High Dimensional Skylines. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 478–495. Springer, Heidelberg (2006)
9. Koudas, N., Li, C., Tung, A.K.H., Vernica, R.: Relaxing join and selection queries. In: VLDB, pp. 199–210 (2006)
10. Jin, W., Morse, M.D., Patel, J.M., Ester, M., Hu, Z.: Evaluating skylines in the presence of equijoins. In: ICDE, pp. 249–260 (2010)
11. Jin, W., Ester, M., Hu, Z., Han, J.: The multi-relational skyline operator. In: ICDE, pp. 1276–1280 (2007)
12. Sun, D., Wu, S., Li, J., Tung, A.K.H.: Skyline-join in distributed databases. In: ICDE Workshops, pp. 176–181 (2008)
13. Vlachou, A., Doulkeridis, C., Polyzotis, N.: Skyline query processing over joins. In: SIGMOD (2011) (to appear)
14. Khalefa, M.E., Mokbel, M.F., Levandoski, J.J.: Prefjoin: An efficient preference-aware join operator. In: ICDE, pp. 995–1006 (2011)
15. Valkanas, G., Papadopoulos, A.N.: Efficient and Adaptive Distributed Skyline Computation. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 24–41. Springer, Heidelberg (2010)
16. Chen, L., Cui, B., Lu, H., Xu, L., Xu, Q.: iSky: Efficient and progressive skyline computing in a structured p2p network. In: ICDCS, pp. 160–167 (2008)
17. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS, pp. 102–113 (2001)
18. Natsev, A., Chang, Y.-C., Smith, J.R., Li, C.-S., Vitter, J.S.: Supporting incremental join queries on ranked inputs. In: VLDB, pp. 281–290 (2001)

Scalable Social Graph Analytics Using the Vertica Analytic Platform

Shilpa Lawande, Lakshmikant Shrinivas,
Rajat Venkatesh, and Stephen Walkauskas

Vertica Systems, An HP Company,
8 Federal St, Billerica, MA 01821, USA
{slawande,lshrinivas,rvenkatesh,swalkauskas}@vertica.com
http://www.vertica.com

Abstract. Social Graph Analytics has become very popular these days, with companies like Zynga, Linkedin, and Facebook seeking to derive the most value from their respective social networks. It is common belief that relational databases are ill-equipped to deal with graph problems, resulting in the use of MapReduce implementations or special purpose graph analysis engines. We challenge this belief by presenting a few use-cases that Vertica has very successfully solved with simple SQL over a high-performance relational database engine.

Keywords: Analytics, Graphs, Data Mining, Vertica, Social Networks, MapReduce, Hadoop, Pig, Influencers, K-core.

1 Introduction

Social networks have become a central feature of our online lives, both as consumers and enterprises. From Farmville to Linkedin to Groupon, many new businesses revolve around knowing your friends and leveraging their collective knowledge, behaviors, opinions and buying power. Indeed, for any data-driven enterprise seeking to provide a personalized and relevant customer experience, it is now no longer just web analytics - effective real-time social network analytics can reap significant rewards. The heart of social network analytics revolves around solving graph problems on large volumes of data at scale and with high performance.

It is a common misconception that relational databases are ill-equipped to deal with graph problems, resulting in the use of custom coded implementations or special purpose graph analysis engines. We challenge this belief by presenting two use-cases that Vertica has very successfully solved with simple SQL over a high-performance relational database engine. The first use-case is to find the influencers in a social graph and to show how it can be used to do A/B testing of products. The second use-case is to solve the problem of counting triangles in a graph and comparing the solutions written in SQL, Hadoop/MapReduce and Pig.

M. Castellanos, U. Dayal, and W. Lehner (Eds.): BIRTE 2011, LNBIP 126, pp. 98–108, 2012.
© Springer-Verlag Berlin Heidelberg 2012

2 Understanding Social Games

In this section, we describe some interesting aspects of social games. Even though these aspects are presented in the context of games, they are quite general in nature, and are relevant to any social network.

2.1 Useful Metrics

Just as investors measure a company's value using a few key numbers, such as earnings or cash flows, the value of a social network can be distilled into a few useful metrics:

- *Churn Rate*: the percentage of users who will stop playing the game in a given month. Some social games have churn rates of 50% per month, or more.
- *Viral Coefficient*: the average organic growth rate in users in a given month. If 100 users are likely to cause five of their friends to join in a given month, that is a viral coefficient of 1.05.
- *Revenue per-user*: the average revenue generated by a user in a month.

A combination of the churn rate and viral coefficient gives the expected user-months per new user, i.e., for each user, the total number of months that the user, plus the people they recruit, play the game. In the case of a 50% churn rate and a viral coefficient of 1.05, each new player is expected to generate 2.10 months of play time[1]. At a viral coefficient of 1.10, that number jumps to 2.22. It is easy to see why a social game company might want to increase the viral coefficient or reduce the churn rate, or both.

2.2 Raising the Viral Coefficient of a Game

In an effort to increase the viral coefficient of a game, a common mistake that a social game company might make would be to advertise in-game actions to all of a user's friends. However, this usually has a negative effect, because those messages are regarded as spam.

To combat this, a strategy that has been successfully used is to differentiate between active and passive users. Active users are those that play the game regularly. Due to the nature of typical incentives in a social game, there is another type of user -- a passive user. Such a user signs up for the game primarily to help a friend out -- for example, the game Farmville [1] rewards users that get their friends to sign up with more farm land. Among active users, advertising in-game actions is effective, whereas passive users tend to not like such messages. Thus, identifying active users and advertising in-game actions to only these users, is a simple yet effective analysis that a social game company might do in order to boost the viral coefficient of their games. This is a problem that can be trivially solved with a relational database. Assuming we have the schema shown in Figure 1, and that the definition of an active user is

[1] The expected user-months per new user is computed as $U_m = 1 + \sum_{n=1}^{\infty}(c, v)^n$, where c is the churn rate, and v is the viral coefficient.

someone who performed at least 5 in-game actions in a month, the current month's active users can be identified with the following query:

```
SELECT u_id FROM activities
WHERE ac_time BETWEEN NOW() - '1 MONTH'::INTERVAL
  AND NOW()
GROUP BY u_id HAVING COUNT(*) >= 5;
```

Table: Activities		
ac_time	timestamp	Time the activity occurred
u_id	integer	User id
what	varchar(50)	Type of activity
it_id	integer	Item id associated with activity (if any)

Table: Items		
item_id	integer	Unique item identifier
name	varchar(30)	Name of item

Table: Interactions		
l_time	timestamp	Time the interaction occurred
src	integer	User id of the originating user
tgt	integer	User id of the target user
type	integer	Type of interaction
content	varchar(1000)	Content of interaction

Table: Active_friends		
src	integer	Source user id
tgt	integer	Target user id

Fig. 1. Example Database Schema

2.3 Finding the Influencers

Apart from finding active and passive users in a social game, it is often useful to find the *influencers* in a social network. An influencer is essentially a user that other users tend to follow. They are also sometimes called mavericks or trend setters. Social networks have been extensively studied since the 1950's and the theory is that influencers can spread information much better than non-influential users.

The term *influencer* by itself, is subjective, and there are several different ways of quantifying whether a particular user (represented by a node in the social network) is an influencer. Some of the metrics that can be used to identify influencers are:

- *Degree* -- is defined as the number of neighbors of a node.
- *Closeness* -- is defined as the inverse of the sum of the shortest distances between the node and every other node in the network.
- *Betweenness* -- is defined as the fraction of shortest paths between any two nodes in the network that pass through the node.

— *Eigenvector* -- is a measure of the importance of the node in the network. If the social network is represented as an adjacency matrix A then the eigenvector is the vector that satisfies the equation $A.x = \lambda.x$, for the greatest value of λ.
— *k-Core* – a sub-graph such that every node is connected to at least k other nodes within the graph.

3 An Example of Social Graph Analysis

In this section, we describe a non-trivial analysis that a social game company might perform, and show how the Vertica Analytic Database can be used to perform the analysis in real time.

The analysis that we want to perform is an A/B test [2] of in-game product placement. Essentially, given a social network, we want to:

1. Identify influencers in the network
2. Give a small subset of influencers product A
3. Give a small random set of users product B
4. Measure the reach of the two products in the network

3.1 Methodology

We simulate this use-case on a social network consisting of a random graph with 90 million nodes, generated using the technique described in [3]. The degree distribution was chosen based on the empirical knowledge that the most common number of friends that a user regularly interacts with in typical social games is between 3 and 5, even though the total number of friends could be a lot higher (we call these *active friends*. In essence, this graph is representative of the social network of recent interactions among users, and has about 450 million edges.

The random graph was then used to generate user interactions over a 30 day time period. These were then loaded into the interactions table in Figure 1. The i_time, type and content columns were randomly generated, while the src and tgt columns were constrained to be neighbors in the random graph. Each 3 day period in the interactions table contains at least one record corresponding to each edge in the random graph. The active friends in past week in the social network can then be identified by the following query:

```
SELECT DISTINCT src, tgt
FROM interactions
WHERE i_time BETWEEN NOW() - '1 WEEK'::INTERVAL
AND NOW();
```

Note that since the time span of the above query is more than 3 days, the output produced by the query will be identical to the generated random graph.

The next step in the process is to identify influencers in the social network. Kitsak et al. [4] showed that the k-Core metric is much better than the degree at identifying

the influence of a node. Coupled with the fact that finding the k-Core of a node is more challenging than the degree, we decided to use that as the measure of influence.

3.2 The K-Core Algorithm

For computing the k-core of the random graph we generated, we used the algorithm described in [5]. The algorithm is iterative in nature and proceeds by deleting all nodes (and the edges incident with them) whose degree is less than the provided k value. This process is repeated until there are no more nodes with degree less than k.

The random graph we generated has roughly 70 million users in the 3-core and no users in the 4-core. In order to find a small enough subset of influential users, we decided to find an approximate 4-core of the graph by running the iteration until the number of nodes fell below a threshold (100,000 in our case).

The SQL queries for computing the approximate 4-core are shown below[2].

Initialization:

```
create table ncore1(src int, tgt int);
create table ncore2(src int, tgt int);
create table lessn (src int, count int);
create table lessn2 (src int, count int);

-- populate the ncore1 table:
insert /*+direct*/ into ncore1 select src, tgt
from active_friends;
```

Iteration (2 iterations shown below):

```
-- populate nodes with degree < 4 into the 'lessn' table
truncate table lessn;
insert into lessn
select src, count(tgt)
from ncore1
group by src
having count(tgt) < 4;

-- populate the 'ncore2' table with the graph
-- obtained by removing all nodes with degree < 4
truncate table ncore2;
insert into ncore2
select * from ncore1
where src not in (select src from lessn)
and tgt not in (select src from lessn);
```

[2] Vertica does not support a stored procedure language, due to which the iteration and control flow logic was performed in an external script.

```
-- repeat the above steps, using 'lessn2' and
-- 'ncore1' as the target tables
truncate table lessn2;
insert into lessn2
select src, count(tgt)
from ncore2
group by src
having count(tgt) < 4;
truncate table ncore1;
insert into ncore1
select * from ncore2
where src not in (select src from lessn2)
and tgt not in (select src from lessn2);
```

At the end of the iteration, the table ncore1 has the nodes remaining in the approximate 4-core of the graph. For the random graph we generated, 8 iterations were enough to reduce the number of nodes to about 34 thousand.

3.3 The A/B Test

Once we identified the influencers, we picked a random subset A of 100 users from the influencers, and a random subset B of 100 users from the entire graph. The users in A were given virtual Coca Cola, and the users in B were given virtual Pepsi. We then simulated the spread of Coca Cola and Pepsi usage using an infection-spreading model [6], according to the following rules:

- If a user's friend is infected with Pepsi or Coca Cola in the current time-step, then the user has a 5% chance of getting infected by the same product in the next time-step
- If a user is infected in the current time-step, they will recover in the next time-step with 100% probability
- If a user was infected by Pepsi or Coca Cola in any previous time-step (but not the current), then they have a 5% chance of getting re-infected by the same product in the next time-step

The simulation was run for 200 time-steps, which generated approximately 20,000 events (where each event was an infection with Pepsi or Coca Cola). We also generated approximately 4.8 billion random events to represent other activities that a user might perform in the game. The 20,000 events of interest (i.e., Coca Cola or Pepsi infections) were interspersed evenly among the 4.8 billion other events. These records were loaded into the activities table (shown in Figure 1) in batches of 400 million records, to simulate how in-game events would be loaded into the database. Then, to measure the reach of Pepsi and Coca Cola, we used the following query:

```
SELECT name, count(*)
FROM activities JOIN items
ON it_id = item_id
```

```
WHERE name IN ('pepsi','coca cola')
GROUP BY name;
```

Indeed the penetration of Coca Cola was higher than that for Pepsi. After 200 time-steps, the number of Coca Cola users was 14976, compared to 5276 Pepsi users.

4 Experimental Evaluation

We ran the example presented in the previous section on a cluster of 4 nodes, where each node had 2 Intel Xeon X5670 processors (12 cores total) and 96GB of RAM. The interactions table had about 1.1 TB of raw data. The in-database size (with 2-way replication) was 366 GB, due to the aggressive compression made possible by columnar storage in Vertica.

Step	Time Taken
Creating a graph of active friends	1 min
Initialization	20 sec
Average time per iteration	30 sec

Fig. 2. K-core Timing

The total time to generate the graph of active friends, and compute the approximate 4-core was about five and a half minutes. Figure 2 shows the timing break-down for each step in the process. The whole process took 8 iterations to reduce the number of users below the threshold (100,000 in our case).

For the second phase of the experiment, each batch of 400 million records (about 10 GB of raw data) was loaded in 15 seconds. Thus, the load rate on 4 nodes, with two-way replication for the activities table was 40 GB per minute. This translates to a rate of 20 TB per hour on a full rack of the HP/Vertica appliance (consisting of 32 nodes). The average query time to find the number of Pepsi and Coca Cola users was about 0.3 seconds, due to two key features in Vertica - the use of run-length encoding (which allowed for very heavy compression) and the ability to evaluate predicates and perform joins on encoded data.

With sub-second response times, such queries can easily be incorporated into a dashboard for an analyst interested in monitoring the spread of the products in real time. Coupled with the fact that the entire process of generating the graph of active friends and running the k-core algorithm took but a few minutes, it is easy to see how the Vertica Analytic Database can be used to perform sophisticated graph analysis in real-time.

5 Comparing Vertica versus MapReduce

A significant number of industry and academic papers tout benefits of using MapReduce to solve large-scale graph analysis problems [7],[8],[9]. One such problem is counting the number of triangles in a graph. This metric can be used to calculate the

clustering coefficient of a graph, which is in turn a useful measure for, among other things, fraud and spam detection.

5.1 Counting Triangles with MapReduce

Let's assume we have an undirected graph with reciprocal edges, so there's always a pair of edges ({e1,e2} and {e2,e1}).

A fast solution for solving this problem when the graph fits on a single node is given by [10] and an approximate method by [11]. Both [12] and [10] present MapReduce solutions to this problem, a simplified version can be summarized as follows. First, do a self join to compute all triads of the graph, by joining the source of one input to the destination of the other. Next, join the open edges of the triads with the original edges. Finally, count all closed triads. Hadoop, a popular MapReduce framework can be made to partition the edges such that distributed joins are relatively straightforward.

Many developers are more comfortable programming in a procedural language and dislike declarative languages such as SQL. Apache Pig, a procedural data flow language for Hadoop, is considered to be superior to Java MapReduce code. The operators exposed by the language and the procedural statement syntax make it straightforward to parallelize Pig programs.

To count triangles in a database, we create an "edges" table and load the graph. Vertica can automate the decision about how to organize storage for the table, specifically the sort order, segmentation, encoding and compression for the data. The algorithm can be expressed as a single SQL query that does a 3-way self-join on the edges table to find triplets of vertices that form a cycle.

Due to lack of space, we do not include the complete solutions here and instead refer the reader to [13] where full source code of the solutions is available.

5.2 Performance Study of Vertica, Hadoop and Pig

We compared the performance of the triangle counting solution on Hadoop, Pig and Vertica. The publicly available LiveJournal social network graph [14] was used to test performance. It was selected because of its public availability and its modest size permitted relatively quick experiments. The modified edges file (in the original file not every edge is reciprocated) contained 86,220,856 edges, about 1.3GB in raw size. We used HDFS dfs.replication=2 (replication=1 performed worse – fewer map jobs were run, almost regardless of the mapreduce.job.maps value). Experiments were run on between 1 and 4 machines each with 96GB of RAM, 12 cores and 10GBit interconnect.

All solutions are manually tuned to obtain the best performance numbers. For the Hadoop and Pig solutions, the number of mappers and reducers as well as the code itself were tweaked to optimize performance. For the Vertica solution, out-of-the-box Vertica is configured to support multiple users; default expectation is 24 concurrent queries for the hardware used. This configuration was tweaked to further increase pipeline parallelism. Figure 3 compares the best performance numbers for each solution.

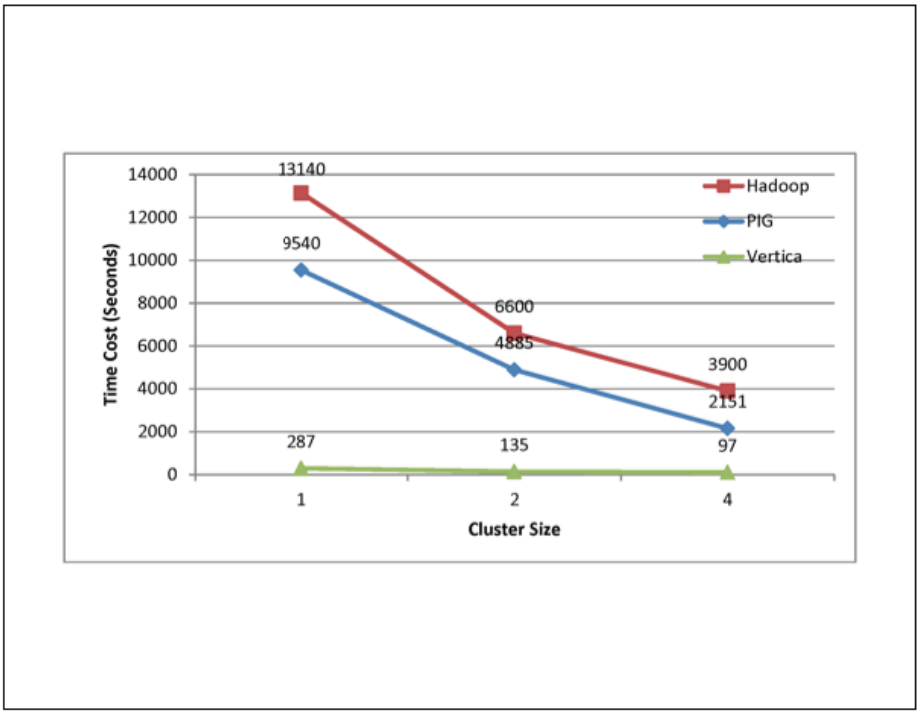


Fig. 3. Triangle Counting Performance Comparison: Hadoop, Vertica, Pig

Pig beat the Hadoop program and a major factor was its superior join performance because it uses hash join. In comparison, the Hadoop solution employs a join method very close to sort merge join. Vertica's performance was much much better – its best time 22x faster than Pig's and 40x faster than the Hadoop program. Even without configuration tweaks, Vertica beat optimized Hadoop and Pig programs by more than a factor of 9x in comparable tests.

There are a few key factors in Vertica's performance advantage:

- Fully pipelined execution in Vertica, compared to a sequence of MR jobs in the Hadoop and Pig solutions, which incurs significant extra I/O. We quantify the differences in how the disk is used among the solutions below in the "disk usage" study.
- Vectorization of expression evaluation and the use of just-in-time code generation in the Vertica engine.
- More efficient memory layout, compared to the frequent Java heap memory allocation and deallocation in Hadoop / Pig.

Finally consider what happens when the use case shifts from counting all of the triangles in a graph to counting (or listing) just the triangles that include a particular vertex. Vertica's projections can be optimized such that looking up all of the edges with a particular vertex is essentially an index search (and once found the associated edges are co-located on disk). On the other hand Pig and Hadoop must process all of the edges to satisfy such a query.

5.3 Disk Usage

Figure 4 shows the peak disk usage at run-time among all 3 solutions in a 4-node cluster (compression was not enabled for Hadoop and Pig because turning it on would reduce disk usage but likely hurt performance). For the input data set of 1.3GB, it takes 560MB to store it in Vertica's compressed storage. In comparison, storing it in HDFS consumes more space than the raw data size. Given the huge differences in disk usage and thus I/O work, along with other advantages outlined above it should come as no surprise that the Vertica solution is much faster.

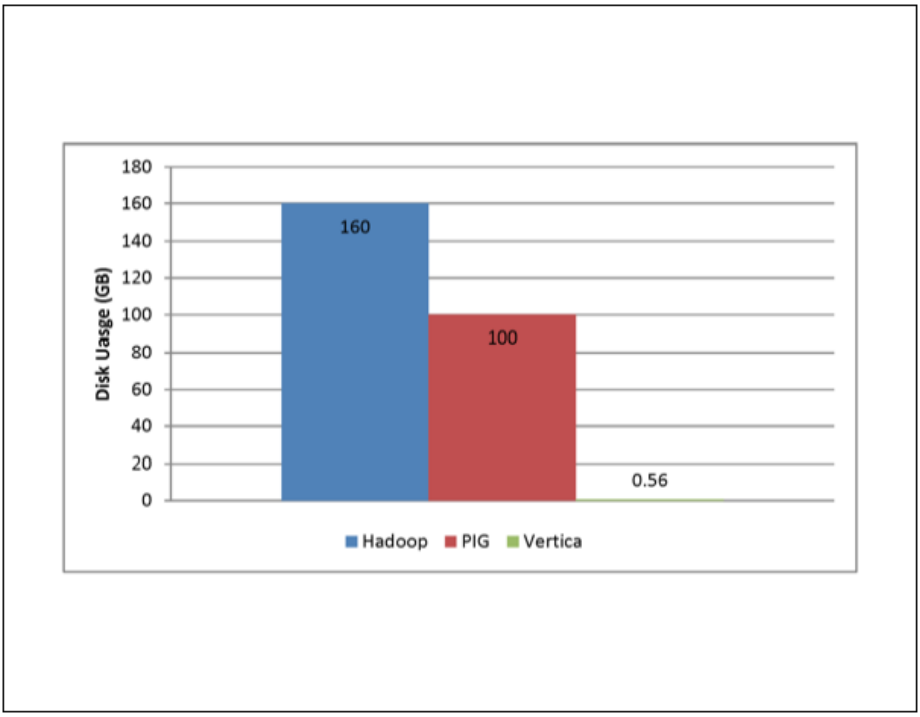


Fig. 4. Disk Usage Comparison: Hadoop, Pig, Vertica

5.4 More on Join Optimization

The Hadoop solution does not optimize for join performance. Both Vertica and Pig were able to take advantage of a relatively small edges table that fit in memory (100s of billions or more edges can fit in memory when distributed over 10s or 100s of machines), with a hash join implementation.

For Pig, the join ordering needs to be explicitly specified. Getting this ordering wrong may carry a significant performance penalty. In our study, the Pig solution with the wrong join ordering is 1.5x slower. The penalty is likely even higher with a larger data set, where the extra disk I/O incurred in join processing can no longer be masked

by sufficient RAM. To further complicate the matter, the optimal join ordering may depend on the input data set (e.g. whether the input graph is dense or not). It is infeasible for users to manually tweak the join ordering before submitting each Pig job.

In comparison, the Vertica columnar optimizer takes care of join ordering as well as many other factors crucial to optimizing for the job run-time.

6 Conclusions

Relational databases can be a powerful tool to perform graph analysis, especially next-generation products like Vertica. Of course not all graph algorithms can be easily expressed in SQL, but several can, and instead of reinventing the wheel, it's often better to use existing products that are perfectly adequate for the problem at hand. We would also like to call the DB community to research efficient database-centric algorithms for computing graph metrics.

References

1. Farmville by Zynga, http://www.farmville.com
2. A/B Testing, http://en.wikipedia.org/wiki/A/Btesting
3. Newman, M.E.J., Watts, D.J., Strogatz, S.H.: Random graph models of social networks. Proceedings of the National Academy of Sciences of the United States of America 99(suppl. 1), 2566–2572 (2002)
4. Kitsak, M., Gallos, L.K., Havlin, S., Liljeros, F., Muchnik, L., Stanley, H.E., Makse, H.A.: Identification of influential spreaders in complex networks. Nature Physics 6(11), 888–893 (2010)
5. Batagelj, V., Zaversnik, M.: An O(m) Algorithm for Cores Decomposition of Networks. CoRR, vol. cs.DS/0310049 (2003)
6. Epidemic Model, http://en.wikipedia.org/wiki/Epidemicmodel#TheSIRSModel
7. PEGASUS - A Petabyte Graph Mining System, http://www.cs.cmu.edu/~ctsourak/pegasusICDM09.pdf
8. Yahoo! Research - Counting Triangles and the Curse of the Last Reducer, http://theory.stanford.edu/%7Esergei/papers/www11-triangles.pdf
9. MapReduce: Simplified Data Processing on Large Clusters, http://labs.google.com/papers/mapreduce-osdi04.pdf
10. Fast Counting of Triangles in Large Real Networks: Algorithms and Laws, http://www.cs.cmu.edu/~ctsourak/tsourICDM08.pdf
11. DOULION: Counting Triangles in Massive Graphs with a Coin, http://www.cs.cmu.edu/~ukang/papers/kdd09.pdf
12. Counting Triangles and the Curse of the Last Reducer, http://theory.stanford.edu/%7Esergei/papers/www11-triangles.pdf
13. https://github.com/vertica/Graph-Analytics-—Triangle-Counting http://snap.stanford.edu/data/soc-LiveJournal1.html

A Near Real-Time Personalization for eCommerce Platform

Amit Rustagi

arustagi@eBay.com

Abstract. In today's competitive environment, you only have a few seconds to help site visitors understand that you have an answer to their needs. How much more powerful would a company's online presence be if the whole experience were relevant to each of the key visitor types and their needs? Personalization has proven its power in helping companies acquire, engage and retain customers by using the technology to accommodate the differences between individuals. Within the e-commerce industry, personalized content relevancy is often thought to lead to improved conversion rates and higher customer satisfaction. People need different things at different stages of their purchase journey. This paper introduces the infrastructure eBay has deployed to drive increased personalization and relevancy to customers.

Keywords: Personalization, Personalization Data Service.
Submission Category: Industry Track.

1 Introduction

Personalization is a process of gathering and storing information about site transactions, analyzing the information, and, based on the analysis, delivering the more relevant information to people who visit Web site. A number of personalization techniques can enable more customized messages for customers in advertisements, product recommendations, news feeds and other marketing tactics. Providing personalization for real-time applications can affect system performance; therefore, the deployment of personalization is important and should be integrated into the overall system design. This is especially true for high-volume Web sites. Web site type should determine selection of personalization techniques. Regardless of type, Web sites look increasingly to the use of personalization to increase repeat business.

It is important to understand that the personalization techniques should be governed by an organization's published privacy policy. While this paper will not go in-depth on the subject, it must be a consideration.

This paper introduces personalization and describes some current techniques. It also explains personalization initiatives at eBay, for bringing more customized and relevant recommendations to customers while still meeting the performance requirements of a high-volume e-commerce site.

M. Castellanos, U. Dayal, and W. Lehner (Eds.): BIRTE 2011, LNBIP 126, pp. 109–117, 2012.
© Springer-Verlag Berlin Heidelberg 2012

1.1 The Evolution of Personalization

Within the e-commerce industry, personalization has gone through different phases. Initially, personalization was used to keep a visitor on a Web site for a longer time, which provided opportunities to advertise and promote products. The next phase offered more expensive or relative products as an attempt to increase average spend. Today, personalization is increasingly used as a means to expedite the delivery of information to a visitor, making a Web site more useful to returning visitors and increasing the number of repeat-customer visits. Companies use different methods to personalize their e-commerce sites. The most common are tailored e-mail alerts, customized content, and account access.

Custom pricing, customized content, targeted marketing, and advertising are more advanced personalization methods that require sophisticated data mining. These methods rely on personalized Web pages and deliver business value by enabling site owners to determine how and when to change site content. However, dynamically building such pages requires additional resources and may affect overall system performance. Minimizing the impact of these pages requires a personalization engine that is scalable to handle a large number of requests, a large and complex content space, and the collection of customers.

Personalization can be a valuable investment that provides valuable services to customers, including more relevant and customized content to meet customer's interests. Within the e-commerce industry, Web sites that incorporate personalization techniques are often considered models for those who want to get started in the space.

1.2 Data Collection and Transaction Profiles

This section introduces current techniques for data collection and the analyzation of transaction profiles to provide customers with a more customized experience. The major steps are transaction data collection, filtering and recommendation development, which may or may not be performed dynamically; part or all of some steps may be performed offline, in batch mode.

The objective of collecting and analyzing transaction information is to develop a profile that describes what might interest a customer in order to provide them with more customized information online. The most common techniques are explicit profiling, implicit profiling, and using legacy data. Explicit profiling asks site visitors to fill out information or questionnaires and allows customers to tell the site directly what they want to see. Implicit profiling captures transaction data such as items viewed and items bought to deliver more relevant information and results to customers. Legacy data is recommended for relevance to a Web site's existing customers and known visitors, because it references past behavior to drive current customization. The techniques can be combined to produce comprehensive profiles. Profile and legacy data become the metadata processed by the filtering techniques. When the profile is available, the next step is to analyze the profile information in order to present or recommend documents, purchases, or actions customized to customers. Making such recommendations can be challenging, as many techniques

for presenting content and making recommendations are in use or under development. Rule-based and filtering techniques are the best known.

2 Personalization at eBay

When consumers find their way to eBay, it's often as the result of an online search they conducted for a specific product, brand, price, or coupon. It's important to capture a potential customer's attention with relevant offers within a short amount of time. Customers increasingly expect a personalized "landing experience", so eBay's goal is to provide more relevant results, customized to their interests.

Fig. 1. Share and leverage combined Personalized data

eBay does see an opportunity to provide more customized information to customers through real-time personalization by using dynamic site and applications to deliver targeted content. Personalization filters out the noise for consumers, helping them avoid information overload. Making such recommendations is the most challenging step.

2.1 Personalization Data Service (PDS)

By most recent activity, behavior, attitude and opportunity, personalization provides strong value proposition to eBay customers because they can more quickly identify items that meet their specific needs. We identified functional needs for the personalization system as a single point of access for Personalization Data (SOA enabled service), ability to easily/quickly expose read only user data captured by offline systems, ability to easily/quickly expose data from new data source, ability to easily/quickly add new dynamic attributes for service users to read/write and support high velocity of change with minimal cost. At the same time, we defined the architectural requirements as a system decoupled from its data sources supporting pluggable data source model, reduce impact on backing data sources by providing caching, standard scaling, availability, robustness requirements and apply support access to read only data that is in a columnar structure.

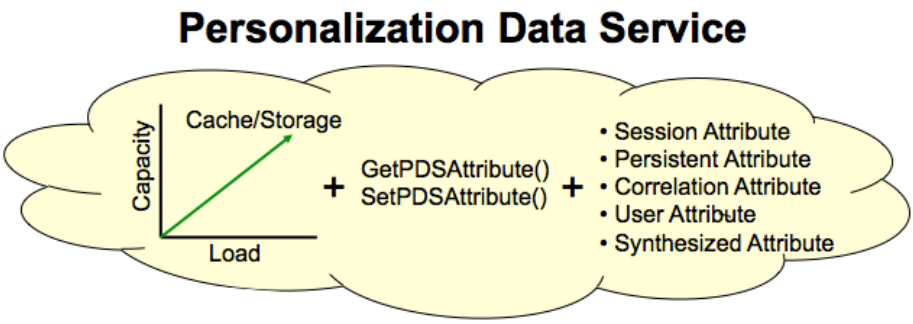


Fig. 2. Universal Service for accessing and storing Personalized data

In order to meet these requirements, Personalization Data Service (PDS) was created, which acts as a central repository of transaction attributes (including transaction activity attributes). It can be used by use cases to evaluate specific personalization rules. It is a system for accessing and storing personalization data. It provides scalable data storage / caching system for storing the personalization data. Also, it offers a common/ simple service interface to access all the personalization data across different data sources.

2.2 Personalization Data Service (PDS) Components

Core components, which make up the Personalization Data Service (PDS), are GemCache, PDS Client/ SOA service, PDS attributes and CacheMesh. GemCache serves as highly scalable, real-time storage and caching system. It's a hybrid of eBay infrastructure, open source and Oracle. It's in memory mySQL database, which uses Oracle for persistent data storage. It has flexible data service interface. It allows for requesting and receiving only the desired PDS attributes by supporting columnar selection. It allows for dynamic creation and access of new PDS attributes via purely metadata configuration. It offers low latency, low app server capacity impact and lower TCO than alternatives. It supports dynamic runtime cluster configurability and manageability. Current eBay (Marketplace) cluster performs more than 7 billion read /writes per day.

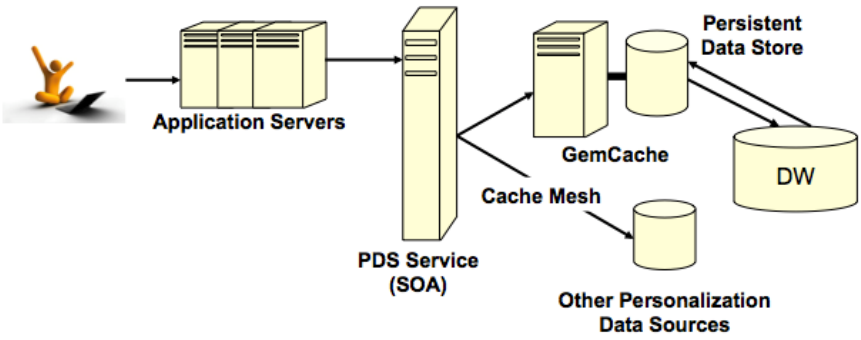


Fig. 3. Personalized Data Service (PDS) system components

CacheMesh inside the PDS service integrates and caches other data sources. It will handle all caching transparently to the client and the adapters. It's the orchestrator in obtaining PDS Attribute values from the data sources.

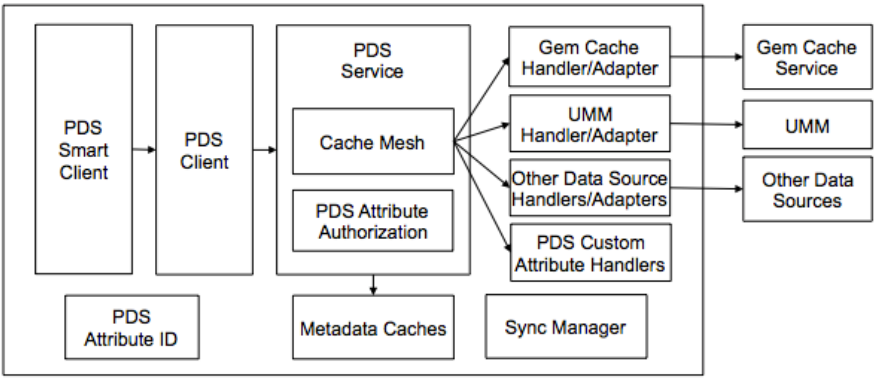


Fig. 4. Personalization Data Service (PDS) architecture

PDS SOA service / PDS attribute objects / PDS Smart Client are used by PDS consumers for getting / setting personalization information. Adapters perform mappings of PDS Attributes to Data Source Attributes. Mappings are configured via metadata in a database (dynamically changeable). Custom Attribute Handlers contain custom mapping java logic. Sync Manager publishes updates of specified data out to registered listeners.

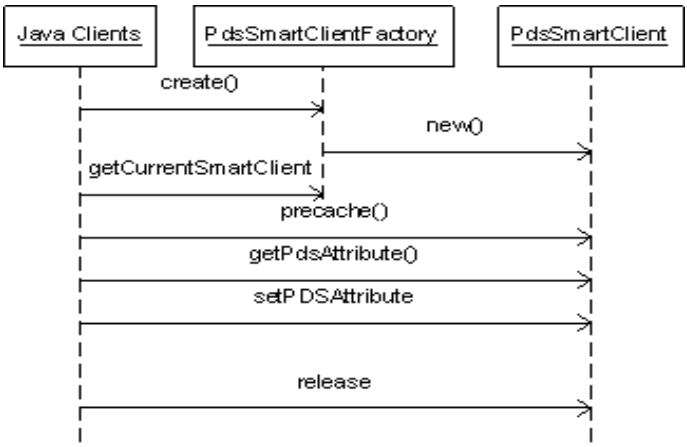


Fig. 5. For integrating with PDS, consumers use personalized Data Service (PDS) SmartClient

Personalization Data Service consumers use the PDS SmartClient for getting/setting the personalization data. PDS SmartClient provides single interface regardless of transport layer and provides sync/async pre cache and release. It supports the learning of attributes used within an application by pre caching the

attributes, which are accessed frequently. It promotes good usage of PDS service by encouraging pre cache and batching of attribute updates. Consumers should see a single getPdsAttributes and setPdsAttributes call made against the PDS Service. PDS consumers create the PdsSmartClient instance through PdsSmartClientFactory; get the current instance from the factoryCall and pre cache to prefetch all required attributes. At this point, get/set calls can be invoked to get/set attributes from/to local map. Finally call release to update all changed attributes to the GemCache.

2.3 Personalization Attributes

At eBay, transaction profiles are a collection of implicit and explicit attributes that we maintain or derive in order to support personalization. There are more than 100 plus personalization attributes that exist today. System supports explicit attributes like age, gender, etc., and implicit attributes like Last Keywords Used, Last Categories Accessed, Last Items Viewed, and Last Items Bought.

A visitor segment is a broad group of site visitors with similar transaction behavior. Segment definitions can be broad (i.e. U.S.-based customers) or more confined (i.e. customers from a specific city with an interest in antiques). Once the attributes and the mechanics of transaction profile are known, rules are developed that more formally define segments. Sample visitor segments might include registered site users who have not purchased any items, customers who have not purchased an item in more than 12 months etc.

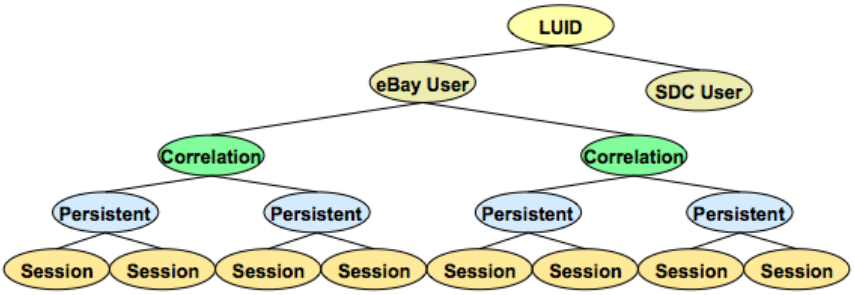


Fig. 6. Attribute hierarchy for storing personalized data

2.4 Leveraging Statistical Analysis/Data Warehouse for Data Mining

When data mining is needed to develop business intelligence and offer highly sophisticated personalization, the processing occurs at the database layer. At eBay, this is accomplished through a joint statistical analysis and data warehouse solution. PDS export job, using ab Initio, transfers certain personalization attributes from GemCache to the data warehouse. Within the data warehouse, set of data elements is identified for modeling, an analytical data set is built in the warehouse where data and tables are integrated, and transformations, aggregations and new predictive variables are derived. Then, analytical data set is extracted into statistical analysis in the model development phase.

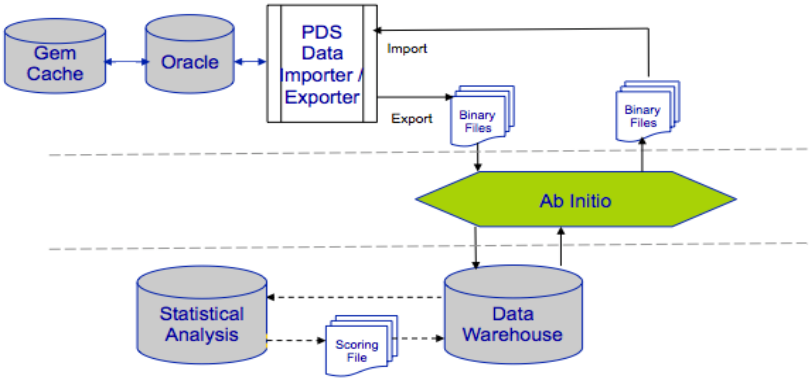


Fig. 7. Personalization Data Service (PDS) integration with Data Warehouse

At this stage, the data is further explored to discover relationships, creating new optimized variable transformations, filtering extreme values and selecting a subset of candidate variables for final modeling. After the model development phase is complete, the production model is ready to be deployed. Statistical analysis for the data warehouse translates the statistical analysis model and registers it as a scoring function in the data warehouse. At that point, PDS import job, using Ab Initio, transfers the personalization attributes from data warehouse to PDS GemCache.

Some of the attributes, which are computed by the scoring process, are User Propensity to Buy, Users Segmentation for a given site and User Rebate/Coupon Sensitivity per site.

2.5 Real-Time Personalization Example at eBay

An eBay customer could be a new user or a returning. A new user comes to eBay.com for the first time and may or may not complete the registration process. Similarly, a returning user has been on eBay in the past and may or may not be a registered user. By understanding the shopping transactions and context, eBay can make recommendation to both types of customers – new as well as returning.

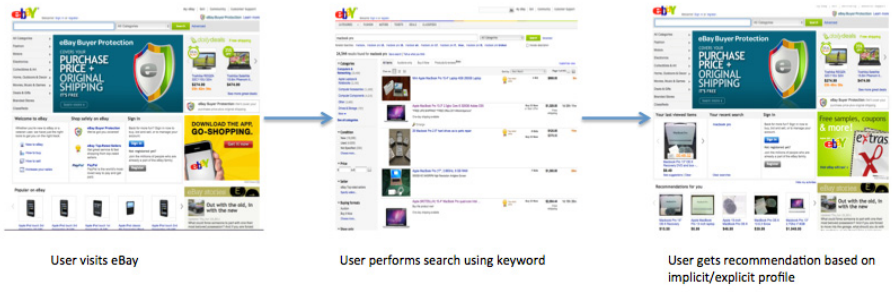


Fig. 8. Recommendations are shown to eBay users as they begin to interact with the site

Customers who come to eBay for the first time have no purchase or behavior history. When these customers visit eBay and search for an item, recommendations are displayed to these users within few seconds leveraging their implicit and explicit profiles. When a returning customer visits eBay, they can be provided with more customized information based on their previous transactions. For example, they can be served related products, recommended products or accessory products that are relevant to their previous interests.

Rule-based techniques provide an environment for specifying business rules to drive personalization. This requires figuring out the appropriate rules. Rule-based techniques can be used with filtering techniques, either before or after the filtering process, to develop the best recommendation. At eBay, rule-based technique is used for cross selling products to customers.

Filtering techniques employ algorithms to analyze Meta data and drive presentation and recommendations. The three most common filtering techniques used are simple filtering, content-based filtering, and collaborative filtering.

Simple filtering relies on predefined groups, or classes, of visitors to determine what content are displayed or what service is provided.

Content-based filtering works by analyzing the content of the objects to form a representation of the visitor's interests. Generally, the analysis needs to identify a set of key attributes for each object and then fill in the attribute values. At eBay, recommending item purchases uses content-based filtering.

Collaborative filtering collects visitors' opinions on a set of objects, using either explicit or implicit ratings, to form like-minded peer groups and then learns from the peer groups to predict other customers' interest in various items. Recommendations produced by collaborative filtering are based on the peer group's response and are not restricted to a simple profile matching. For product recommendations, collaborative filtering is most suitable for homogeneous, simple products, such as consumer electronics.

2.6 Improving Personalization Efficiency

eBay uses technologies and techniques for increasing cost-efficiency of serving personalized data. Performance is maximized when cache hits occur close to the browsers. Similarly, more complex and sophisticated personalization techniques are introduced, as it gets closer to the database layer.

When database changes arrive rapidly, as they do during a purchase, a trigger monitor can be implemented to watch changes. Changes can then be propagated forward from the database server to the browser. When a certain number of changes, or certain changes, occur, the trigger monitor rebuilds the affected Web pages and distributes updated pages to the caches. This technique ensures data is current and performance is maximized, making it appropriate for use with dynamic personalized pages and ideally reducing a page's only dynamic content.

The trigger monitor is the key technology at the heart of a robust implementation of intelligent content distribution. Our site uses an integrated cache to serve dynamic pages. An externalized API enables the server to load and invalidate pages as needed.

A trigger monitor keeps caches current while content is changing rapidly. Sites can benefit from content caches, as well as the trigger monitor.

The cost of personalization can be reduced by the degree of personalization. For example, instead of creating pages specialized for each individual client, sets of pages specialized (tagged) to groups of visitors could be created. This could significantly reduce the total number of pages and allow reuse of some pages, thus increasing the utility of caching. This reduced level of personalization can be provided at a content cache. You can also vary the degree of personalization based on server load. When servers are heavily loaded, the amount of personalization could be minimized.

3 Privacy Considerations

As mentioned earlier, the foundation for any personalization program must consider customer privacy. Privacy regulations vary by country and organizations should consult their legal or privacy teams for application.

4 Results and Conclusion

Dynamic personalization at eBay drove an increase in click-through rates for home page (www.ebay.com) and Sign Out Placements.

Personalization requires analysis of business goals and the development of business requirements, use cases, and metrics. Once these are fully understood, organizations may find that their personalization strategies don't require substantial augmentation of their application environments.

Organizations should also examine how their needs are likely to change and whether their approach to personalization will enable them to move along the continuum from preliminary targeting efforts to tapping the potential of every possible segment.

References

1. Web site personalization by Willy Chiu, `http://www.ibm.com/developerworks/websphere/library/techarticles/hipods/personalize.html`
2. MySQL in eBay's Personalization Platform, `http://www.mysqlconf.com/mysql2008/public/schedule/detail/1240`
3. Forrester Research, Inc., Smart Personalization (July 1999)

Author Index

Barber, Ronald 1
Bendel, Peter 1
Blakeley, José A. 53
Böse, Joos-Hendrik 38

Chen, Qiming 23
Czech, Marco 1

Dittrich, Jens 65
Draese, Oliver 1
Dyke, Paul A. 53

Galindo-Legaria, César 53

Ho, Frederick 1
Hrle, Namik 1
Hsu, Meichun 23
Hübner, Florian 38

Idreos, Stratos 1

James, Nicole 53
Jindal, Alekh 65

Kim, Min-Soo 1
Kleinerman, Christian 53
Koeth, Oliver 1
Krüger, Jens 38

Lawande, Shilpa 98
Lee, Jae-Gil 1
Li, Tianchao Tim 1
Lohman, Guy 1

Morfonios, Konstantinos 1
Murthy, Keshava 1

Peebles, Matt 53
Plattner, Hasso 38

Qiao, Lin 1

Raghavan, Venkatesh 81
Raman, Vijayshankar 1
Rundensteiner, Elke A. 81
Rustagi, Amit 109

Shrinivas, Lakshmikant 98
Sidle, Richard 1
Srivastava, Shweta 81
Stolze, Knut 1
Szabo, Sandor 1

Tkachuk, Richard 53
Tosun, Cafer 38

Venkatesh, Rajat 98

Walkauskas, Stephen 98
Washington, Vaughn 53
Wu, Ren 23

Zeier, Alexander 38