Fourth International Workshop on Storage Network Architecture and Parallel I/Os

On RDBMS-Integrated Disk-Based Architecture for Managing Massive Dormant Data in a Compressed Format

Miroslav Dzakovic
Intel Corporation
miroslav.dzakovic@intel.com

Charles M. Ellison
University of Illinois
celliso2@uiuc.edu

Abstract

Managing dormant data in very large databases (VLDB) is about striking a balance between cost and performance. While magnetic tapes have long been a standard answer, the downsides limit the applicability for all VLDB. This paper describes a class of engineering and scientific data management applications that need a cost-effective solution with better support for updating and lower granularity of data access. To meet the needs of such applications, it then introduces an architectural extension to RDBMS. In the architecture, data is compressed using custom compression techniques, stored outside an RDBMS, and made available through views using SQL Table functions. This architecture is applicable to many industrial RDBMS, and it offers both linear speed-up and scale-out. The paper then discusses preliminary findings from a prototype implementation on Oracle RDBMS.

1. Introduction

The internal organization of data in Very Large Databases (VLDB) is oriented toward recognizing and utilizing data usage patterns. More frequently used data is typically stored on high-performance high-cost media, while less frequently used data is stored on low-performance low-cost media. By separating dormant and active data, one not only saves cost, but also improves performance of active data—as separate media is used for dormant data, the high-performance media is no longer clogged with dormant data [7].

The traditional industry approach for managing dormant data in very large databases is to use a tertiary storage consisting of media juke box with magnetic tapes or optical disks. Such tertiary storage systems scale in size very well and offer automated access to the data cheaper than disk-based systems. However, these solutions also require a clear distinction to be made between active and dormant data, which can limit the applicability of the platform.

ObservationID	TxnID	AttributeID	Value
2641	5120314	126	3.2
2641	5120314	128	1.7
2641	5120314	131	405
752	5120314	127	1256
752	5120314	128	0.9
...	...	...	...
2641	5178255	226	1.2
2641	5178255	421	0.17
2135	5178255	226	1.2

Figure 1. Table `OBSERVATION_ATTRIBUTE`

In this paper we first describe a class of engineering and scientific data applications that need better support for managing dormant data. Next, we propose a disk-based architectural solution that directly addresses these difficulties. We take a closer look at the proposed architecture, focusing on most important issues, namely data organization, scalability, and integration with RDBMS. We also present our preliminary findings and discuss our concerns with the direction.

2. Engineering and Scientific Data Needs

Data in many engineering and scientific databases is used for various types of statistical analyses. Data for statistical analysis typically consists of large number of *observations*, where each observation has several *feature attributes*.

Table `OBSERVATION_ATTRIBUTE` shown in figure 1 is an example of a normalized table containing atomic-level data. One record in the table represents one attribute value of a given observation, collected as part of some transaction. A transaction is assumed to take place at a given point in time, and it collects data for multiple observations at the time. Attributes of observations are collected across many transactions (as is the case for Observation 2641).

When organizing data, a database architect would consider data distribution, usage frequency of observations, as

0-7695-3097-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SNAPI.2007.16

Authorized licensed use limited to: IEEE Xplore. Downloaded on December 4, 2008 at 10:37 from IEEE Xplore. Restrictions apply.

well as relevance of attributes. In the interest of reducing cost and improving performance, one would tend to treat data for most frequently used observations and their most relevant attributes as active data; other data would be considered dormant. However, such a distinction is not easily made a priori.

Suppose that there is a yield problem in manufacturing. Many units are failing test and getting scrapped, and our goal is to find the root cause of the low yield. The basic principle of commonality analysis requires that we use passing units' data in addition to failing units' data. As the volume of such data can be prohibitive for complex statistical analysis, initial data analysis can be performed on a sample of observation. The sampled set of observations has to be chosen using a non-biased random sampling. In addition, it is likely to be performed using attributes that are considered most relevant. Should we fail to find the relevant signal in the attributes we initially chose, we need to use a larger set of attributes.

We derive two important characteristics of this data usage:

- Extracted observations originate from many different time points. This stems from the fact that extracted data set is very large, or, if it is sampled, it is taken from random points in time. Otherwise, the sample data set will be biased toward those time periods that are more significantly represented in the data set.
- At any point in time, any combination of attributes (or all of them) could be used for analysis.

Finally, we would like to briefly comment on the need to update data chunks. Many engineering and scientific data sets have a need for updating. For example, in manufacturing test data, a set of test results may need to be marked obsolete or updated with more recent results when the tests are repeated.

Traditional tape systems have relatively large segments, and such organization would require updating month-old chunks of unit data. Moreover, updating is typically handled by "blanking" out entire segments and creating a newly updated copy in a different place on the media, even if only a single byte has to be changed. A naïve approach (even with deferred update policy) can lead to a huge waste of space, given that smallest segments have at least several MB. In summary, the traditional approach does not lend itself well to organizing data by a non-time attribute, and does not support small updateable chunks of data.

3. Proposed Architecture

Figure 2 shows a high level diagram of the proposed architecture, which has two tiers. Tier-1 is a standard RDBMS (such as Oracle) that manages active data. Tier-2 consists of an arbitrary number of slave nodes that manage data in compressed format. The slave nodes communicate only to a daemon which interfaces between slave nodes and the Tier-1 RDBMS itself. Tier-1 RDBMS does all the SQL processing by combining the Tier-1 and Tier-2 data.

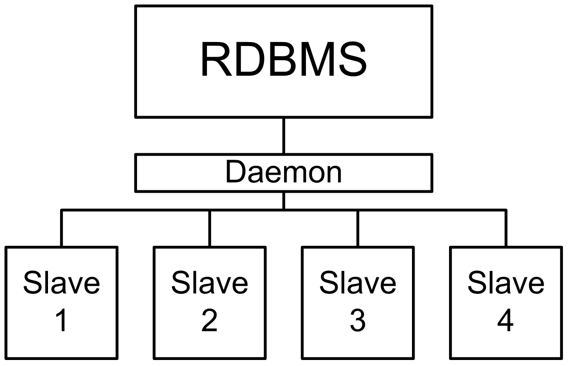

Figure 2. Architecture

Only the largest tables need to be stored on Tier-2. Tier-1 (RDBMS) stores the complement of schema tables and the actively used subset of the Tier-2 data. When a full range of data is needed (both active and dormant), only Tier-2 tables are used and there is no need to union data from multiple Tiers. This approach also allows Tier-1 to be used for deferred updates to Tier-2 and to act like the "redo log" system for Tier-2. If the insertion process into Tier-2 fails, the data still resides on Tier-1 and simply has to be reinserted into Tier-2.

Many tertiary storage systems have been designed to successfully integrate with RDBMS—they feed data into RDBMS just as a remote database would do, and RDBMS in turn does complex query processing [4]. Hybrid architectures have been explored in Object-Relational DBMS using external functions [8, 11]. Particularly interesting for us are the SQL Table Functions [12], which allow an external function to accept parameters and return a set of rows that SQL interprets as a table. In our proposed architecture, we follow this latter approach to integrate externally stored and compressed data with RDBMS.

To illustrate data organization in our architecture, we shall use the table `OBSERVATION_ATTRIBUTE` as shown in figure 1. Suppose that we want to organize data in chunks by ObservationID and TxnID. We first create a small table on Tier-1 that will map each ObservationID and TxnID combination into a newly generated unique key, as shown in figure 3. The rest of the data from the original table is available on Tier-2. The new column Key contains a key value that can be used to return the set of rows from Tier-2. The original table is reconstructed with a view as shown in figure 4. The external function `cc_ext_oa_key` is an SQL Table function. It returns an object having a set of rows for a

Authorized licensed use limited to: IEEE Xplore. Downloaded on December 4, 2008 at 10:37 from IEEE Xplore. Restrictions apply.

ObservationID	TxnID	Key
2641	5120314	253911236
752	5120314	253911237
...		
2641	5178255	254016442
2135	5178255	254016443

Figure 3. Table OBS_ATTR_PSEUDOKEY

```
CREATE VIEW OBSERVATION_ATTRIBUTE_2 AS
  SELECT
    oa.ObservationID, oa.TxnID,
    cc.AttributeID, cc.Value
  FROM
    OBS_ATTR_PSEUDOKEY oa,
    TABLE(cc_ext_oa_key(oa.key)) cc;
```

Figure 4. View OBSERVATION_ATTRIBUTE_2

given key value, essentially performing a nested loops join. The `TABLE()` operator converts the rows into a table with columns AttributeID and Value.

The proposed architecture answers the challenges of engineering and scientific data needs because it organizes data into relatively small updateable chunks (a few to a few hundred attributes). We now take a closer look at two main components of the architecture, the daemon and the slaves, and we reflect on data compression, which is a key factor to reducing the cost.

3.1. Daemon

The daemon is a lightweight process with a small memory footprint. Its main task is to provide a level of indirection for communication between RDBMS and slaves. The daemon keeps a very small bucket map (64K entries) that distributes hash values across slave nodes, akin to Teradata [5]. Apart from computing the hash value and routing the data, the daemon performs no other function. This ensures minimal impact to Tier-1 resources.

The process architecture of the daemon in our prototype has one thread for each client, and one thread for each slave. In order to reduce the overhead of creating pipes and threads, the connection with a client is kept open for the duration of the SQL session. In our prototype, Oracle9i on Windows 2000 helps with this automatically—at the first invocation of the external function, a new process (extproc) is created and attached to the function DLL. Static variables are used inside the DLL function in order to retain the references to the pipes between function calls.

3.2. Slaves

Slaves extract and insert data for a given combination of key, table name and partition name. Internally, each slave uses a hash value of key, table name and partition name to store and retrieve data chunks. The chunks are kept in spans which are in turn kept in OS level files. We developed a system for managing the spans, which for most insertions only require a single disk read and a single disk write.

Each slave is a single threaded process in our prototype. This makes maintaining consistency easy, since we do not have to worry about locking issues with concurrent reads/writes. Scaling-out the system increases performance. This is because slaves act in parallel and share no data, so additional slaves provide additional performance.

As mentioned before, the Tier-1 serves as a redo log for ensuring persistence and consistency of Tier-2 data.

3.3. Compression

Many industry solutions have implemented a variety of compression techniques inside RDBMS [13]. RDBMS typically apply some lightweight data compressions, and some tape drives in industry apply compression that is about half as effective the standard gzip compression [9]. In these cases, factors such as throughput time [2] and CPU overhead impose limitations on applicable compression techniques. Because the cost and not the performance is of main concern in case of Tier-2 systems, we favor the compression ratio over other factors, and we stretch it at the expense of CPU resources on slave nodes.

The compression in our system consists of two parts—semantic packing and general-purpose compression. Packing is based on inside knowledge of the data. It removes redundancy before we apply general purpose compression. Packing spends a lot of effort looking for known patterns in the data, based on data profiles collected previously. The level of trade-off between writing custom packing routines and overall space savings is entirely up to the user of our architecture.

As an example of custom packing, we noticed that in one table, three of the fields almost always contained the same values in each record. We decided to represent those three fields containing those three specific values using one bit. If the fields were different, the bit would be flipped the other way, and the differing data would follow. A similar technique was used for wide, fixed length fields, where the majority of data actually occupies a narrow region. As an extreme example, imagine a 32-bit integer field where the majority of the values fall between zero and three.

We use publicly available zlib to do in-memory general-purpose compression and decompression. It works well on small pieces of data—in our case, sometimes as small as

Authorized licensed use limited to: IEEE Xplore. Downloaded on December 4, 2008 at 10:37 from IEEE Xplore. Restrictions apply.

fifty bytes—because it does not need a large header. This combination of data-specific packing and general-purpose compression produces compression ratios that are better than those in the traditional systems.

4. Preliminary findings and Discussion

We demonstrated the feasibility of the architecture by implementing a prototype using Oracle9i RDBMS. In this section we discuss early concerns and some preliminary findings from the prototype.

4.1. Performance

We found that extracting data in a serial single-user mode from Tier-2 system with a single slave node is about three times slower than extracting data from equivalent Tier-1 data. Table 1 shows performance comparison on a sample of 5 typical queries. The relative numbers (ratio) are most relevant.

Table 1. Elapsed time for sample queries for single and two-tier systems

Test	Single Tier (s)	Two Tiers (s)	Ratio
1	4.06	4.28	1.054
2	6.99	16.96	2.426
3	7.91	21.98	2.779
4	11.7	33.32	2.848
5	15.07	46.48	3.084

4.2. Scalability

Our proposed architecture offers the ability to scale out, due to its shared-nothing parallelism. Since none of the slaves need to communicate with each other and can perform entirely independently and in parallel, adding additional slaves improves performance proportionally. The daemon is the only potential bottleneck, which was the reason for making it very lightweight. Additionally, the architecture offers the ability to balance processing and storage capacity according to one's needs, through the use of custom compression. This is another advantage over traditional tertiary storage systems.

Our prototype system, implemented using 32-bit Intel Architecture, supports up to 65536 slaves, each of which can handle at least 12 TB of compressed data.

4.3. Cost Efficiency

While we do not offer a detailed cost comparison due to many soft and unknown costs, we do offer the following observations.

- The cost-difference between tape-based systems and disk-based systems is often quoted to be an order of magnitude as an argument in favor of tertiary storage. However, high-capacity disks have become identified as an option for mass storage systems in general [6]. Many industry solutions for aged data incorporate a storage hierarchy [3], and sometimes rely exclusively on disks for data storage, and use tapes only for backup and compliance [1, 10].

- The actual compression factor achieved with our custom compression solution was more than 12x, which is four times better than Oracle compression (table and index). Our tests were done on a mix of real-world data. Assuming a price of $14/GB for low-cost disk storage of 1 PB of data, we would further reduce the disk storage cost by $3.6 M compared to Oracle compression.

4.4. Impact on Tier-1

Compared to running equivalent queries on Tier-1 data only, a significant increase in Tier-1 CPU time (2x) is noted for Tier-2 queries. Table 2 shows the CPU consumption on just the RDBMS component of the Tier-1 system. This data suggests that there is a large overhead involved with the SQL Table function or external function calls themselves. In production, the sessions that are using Tier-2 data could be given lower priority than pure Tier-1 sessions, which would limit the CPU impact on time-critical queries.

Table 2. CPU consumption on RDBMS for single and two-tier systems

Test	Single Tier (s)	Two Tiers (s)	Ratio
1	2.56	2.37	0.926
2	5.79	9.34	1.613
3	6.03	10.81	1.793
4	10.04	18.03	1.796
5	12.73	24.03	1.888

Incidentally, we found that if a remote query is used in place of the external function call, the CPU overhead is negligible in Oracle. We would like to see the CPU overhead of external function calls reduced to the comparable level.

Authorized licensed use limited to: IEEE Xplore. Downloaded on December 4, 2008 at 10:37 from IEEE Xplore. Restrictions apply.

5. Further Research

We identify some of the major directions for further research:

- As mentioned before, there is a large CPU overhead using the SQL Table function in Oracle9i. We would like to see RDBMS vendors implement support for light-weight communication with SQL Table functions.
- Latency to the slave nodes needs to be reduced. As a workaround to the latency problem, the look-ahead of slave keys can be implemented.
- We propose and would like to see an attempt to apply this architecture to medical health records and other databases with a similar problem statement.

6. Summary

Many scientific and engineering data management applications need a cost-effective solution for managing dormant data. While tape media has long been a platform of choice, it limits the granularity of data access and ability to store data by non-time related attributes.

We described a class of applications that need both better updatability and lower granularity of data access. Next, we proposed an architecture as an answer to this problem.

Our architecture consists of a shared-nothing layer of slave nodes (which manage data and do compression and decompression), a set of external functions that are used as SQL table functions, and a daemon that acts as an intermediary between the slaves and the Tier-1 RDBMS. The architecture is inherently scalable and maintains data consistency. The architecture limits impact to the Tier-1 system because the heaviest data operations (compression and IO) are performed by the slave nodes.

Data is compressed using both general-purpose compression routines and data-specific packing to maximize overall compression ratio. This technique showed compression ratios four times larger than built-in RDBMS compression in Oracle.

Our proposed approach enables using small chunk sizes, which would be too small for a tape-based tertiary storage. Re-writing (updating) data in our architecture is straightforward (identical to inserts), and the small chunk size benefits both reads and writes.

We built a working prototype of the proposed architecture using industry-standard RDBMS (Oracle9i on Windows 2000) and relatively cheap hardware. Our preliminary findings attest to the feasibility of the architecture, and bring to light some directions for improvements and further research.

References

[1] Storhouse with maid technology. Available on: `http://www.filetek.com/software/product_sheets/datasheet_sthmaid.pdf`, 2004.

[2] Z. Ben-Miled, H. Li, O. A. Bukhres, M. Bem, R. Jones, and R. J. Oppelt. Data compression in a pharmaceutical drug candidate database. *Informatica (Slovenia)*, 27(2):213–224, 2003.

[3] D. I. Boomer. Relational database active tablespace archives using hsm technology. Available on: `http://www.hpss-collaboration.org/hpss/about/BoomerRDBMSHSM.pdf`.

[4] J. Felipe Carino, P. Kostamaa, A. Kaufmann, and J. Burgess. Storhouse metanoia - new applications for database, storage & data warehousing. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 521–531, New York, NY, USA, 2001. ACM Press.

[5] J. Gray. Gray's notes on parallel database systems. Available on: `http://www.research.microsoft.com/~gray/PDB95.doc`, 1995.

[6] A. Guha. A new approach to disk-based scalable mass storage system. Available on: `http://romulus.gsfc.nasa.gov/msst/conf2004/Papers/MSST2004-47-Guha-a.pdf`, 2004.

[7] W. H. Inmon. Managing the lifecycle of data. Available on: `http://www.sun.com/solutions/documents/white-papers/bidw_ManagingTheLifecyleOfData.pdf`, 2005.

[8] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational dbms. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 379–389, New York, NY, USA, 1998. ACM Press.

[9] T. Johnson and E. L. Miller. Performance measurements of tertiary storage devices. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 50–61, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[10] J. Li, W. keng Liao, A. Choudhary, and et al. Parallel netcdf: A high-performance scientific i/o interface. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 39, Washington, DC, USA, 2003. IEEE Computer Society.

[11] T. Mayr and P. Seshadri. Client-site query extensions. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 347–358, New York, NY, USA, 1999. ACM Press.

[12] B. Reinwald, H. Pirahesh, G. Krishnamoorthy, G. Lapis, B. Tran, and S. Vora. Heterogeneous query processing through sql table functions. *International Conference on Data Engineering*, 00:366, 1999.

[13] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 59, Washington, DC, USA, 2006. IEEE Computer Society.

Authorized licensed use limited to: IEEE Xplore. Downloaded on December 4, 2008 at 10:37 from IEEE Xplore. Restrictions apply.