A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database

Hasso Plattner
Hasso Plattner Institute for IT Systems Engineering
University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
hasso.plattner@hpi.uni-potsdam.de

Categories and Subject Descriptors

H.2.0 [**Information Systems**]: DATABASE MANAGE-MENT—*General*

General Terms

Design, Performance

1. INTRODUCTION

Relational database systems have been the backbone of business applications for more than 20 years. We promised to provide companies with a management information system that covers the core applications, including financials, sales, order fulfillment, manufacturing, as well as human resources, which run from planning through business processes to individually defined analytics. However, we fell short of achieving this goal. The more complex business requirements became, the more we focused on the so-called transactional processing part and designed the database structures accordingly. These systems are called OLTP (Online Transactional Processing) systems. Analytical and financial planning applications were increasingly moved out to separate systems for more flexibility and better performance. These systems are called OLAP (Online Analytical Processing) systems. In reality, parts of the planning process were even moved off to specialized applications mainly around spreadsheets.

Both systems, OLTP and OLAP, are based on the relational theory but using different technical approaches [13]. For OLTP systems, tuples are arranged in rows which are stored in blocks. The blocks reside on disk and are cached in main memory in the database server. Sophisticated indexing allows fast access to single tuples, however access gets increasingly slower as the number of requested tuples increases. For OLAP systems, in contrast, data is often organized in star schemas, where a popular optimization is to compress attributes (columns) with the help of dictionaries.

After the conversion of attributes into integers, processing becomes faster. More recently, the use of column store databases for analytics has become quite popular. Dictionary compression on the database level and reading only those columns necessary to process a query speed up query processing significantly in the column store case.

I always believed the introduction of so-called data warehouses was a compromise. The flexibility and speed we gained had to be paid for with the additional management of extracting, and loading data, as well as controlling the redundancy. For many years, the discussion seemed to be closed and enterprise data was split into OLTP and OLAP [9]. OLTP is the necessary prerequisite for OLAP, however only with OLAP companies are able to understand their business and come to conclusions about how to steer and change course. When planned data and actual data are matched, business becomes transparent and decisions can be made. While centralized warehouses also handle the integration of data from many sources, it is still desirable to have OLTP and OLAP capabilities in one system which could make both components more valuable to their users.

Over the last 20 years, Moore's law enabled us to let the enterprise system grow both in functionality and volume [16]. When the processor clock speed hit the 3 GHz level (2002) and further progress seemed to be distant, two developments helped out: unprecedented growth of main memory and massive parallelism through blade computing and multi-core CPUs [14]. While main memory was always welcome for e.g. caching and a large number of CPUs could be used for application servers, the database systems for OLTP where not ideally suited for massive parallelism and stayed on SMP (symmetric multi processing) servers. The reasons were temporary locking of data storage segments for updates and the potential of deadlocks while updating multiple tables in parallel transactions. This is the main reason why for example R/3 from SAP ran all update transactions in a single thread and relied heavily on row level locking and super fast communication between parallel database processes on SMP machines. Some of the shortcomings could be overcome later by a better application design, but the separation of OLTP and OLAP remained unchallenged.

Early tests at SAP and HPI with in-memory databases of the relational type based on row storage did not show significant advantages over leading RDBMSs with equivalent memory for caching. Here, the alternative idea to investigate the advantages of using column store databases for OLTP was born. Column storage was successfully used for

many years in OLAP and really surged when main memory became abundant [20, 5].

At the HPI, two and a half years ago, we started to analyze whether it is feasible or not to perform OLTP operations on an in-memory column store database. Many bachelor, master, and PhD projects focused on this topic. In the following paragraphs I would like to discuss our findings. Some sections are disputing common beliefs, others discussing options for future developments.

2. COLUMN STORAGE IS BEST SUITED FOR MODERN CPUS

Modern multi-core CPUs provide an enormous amount of computing power. Blades with 8 CPUs and 16 cores per CPU will populate next-generation blade servers. That gives us 128 computing units with up to approximately 500 GB of main memory. To optimize the use of these computing devices we have to understand memory hierarchies, cache sizes, and how to enable parallel processing within one program [6]. We consider the memory situation first. Enterprise applications are to a large extent memory bound, that means the program execution time is proportional to the amount of memory accessed for read and write operations or memory being moved.

As an example, we compare a full table scan of SAP's accounting document line items table, which has 160 attributes, in order to calculate a total value over all tuples. In an experiment we did with 5 years worth of accounting of a German brewery, the number of tuples in this table was 34 million. In the underlying row database, 1 million tuples of this particular table consume about 1 GB of space. The size of the table was thus 35 GB. The equivalent column store table size was only 8 GB because of the more efficient vertical compression along columns. If we consider that in real world applications only 10% of the attributes of a single table are typically used in one SQL-statement (see Figure 1), that means for the column store at most 800 MB of data have to be accessed to calculate the total values [1]. Figure 2 shows (schematically) that the row store with horizontal compression cannot compete if processing is set-oriented and requires column operations. Even with the appropriate index the amount of data accessed is orders of magnitude higher.

SELECT c1, c4, c6 FROM table WHERE c4 < ?

Figure 1: Example Query and Schema

According to our analyses of real systems with customer data, most applications in enterprise computing are actually based on set processing and not direct tuple access. Thus, the benefit of having data arranged in a column store is
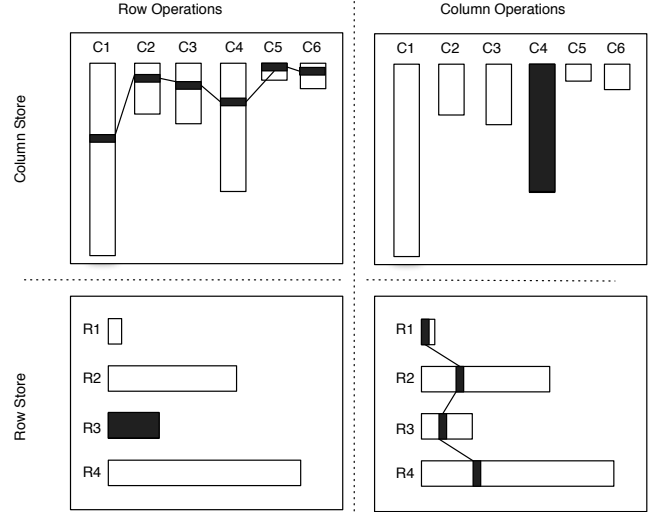

Figure 2: Data Access in Row- and Column Storage

substantial. In addition to this, we can execute most of the calculations on row level using the compressed, integer format. We see here a performance gain of a factor 100 - 1000 in comparison to the same calculation executed on non-compressed data formats at the application level. The application layer has to work with minimal projections in local SQL statements and avoid using more generic SQL statements in subroutines to support the reduction in memory access.

On top of these benefits comes the introduction of parallel processing. According to Hennessy, the difficulty of creating parallel processing programs is to break up a program into equal-sized pieces, which then can be processed in parallel without much synchronization [12]. The scan operation through one or more columns is exactly what we are looking for. This operation can indeed be split easily into equal parts and distributed to multiple cores. The standard operations of OLAP engines and any other formal application logic, e.g. calculation of due dates, currency conversion, working days for a given date interval etc., can be handled by stored procedures operating on the integer values of the compressed columns.

All calculations on tuple level will automatically be parallelized, since they are completely independent of each other. The first level of an aggregation will be executed synchronously for each qualified tuple. The synchronization between the core processes is minimal. Further aggregation along given hierarchies takes place as a second step on the accumulated data. The same applies to sorting by attributes or sequencing by time.

Even if only a few tuples match the selection predicate, the introduction of indices is not necessary because the scanning speed is so enormous, especially if parallel processing across multiple cores is active. On current CPUs, we can expect to process 1 MB per ms and with parallel processing on 16 cores more than 10MB per ms. To put this into context, to look for a single dimension (i.e. attribute) compressed in 4 bytes, we can scan 2.5 million tuples for qualification in 1 ms. With this speed in mind, we will not even provide a primary key index for most of the tables anymore but use the full column scan instead. Column storage is so well

suited for modern CPUs that the full scope of the relational algebra can be used without shortcomings in performance. It is important to note that every attribute now represents a potential index. There are no restrictions anymore for the applications to select data only along on predefined navigation paths. The delegation of most of the calculations to the database layer cleans up the application layer and leads to a better separation of concerns. This will result in a higher quality of programs and allow a better lifecycle with ongoing development. The hard disk is used only for transaction logging and snapshots for fast recovery. In fact, disk has become yesterday's tape [11].

3. CLAIM:
COLUMN STORAGE IS SUITED FOR UPDATE-INTENSIVE APPLICATIONS

Column store databases are said to be expensive to update [8]. Having all data in main memory greatly improves the update performance of column stores, but we still have to consider the potential expansion of the attribute dictionaries, which could lead to a situation where the compression has to be recalculated and thus affects the entire column. Therefore, we analyzed the updates in a financial system (Figure 3) in more detail.

3.1 History of SAP's Database Table Design

The large number of materialized views and materialized aggregates might be astonishing at first glance. This redundancy became necessary to achieve reasonable response times for displaying the line items and account totals. The higher number of inserts and the problematic updates of redundant data using database triggers or procedural code was the price to pay. The customer-defined roll-ups into cubes in the OLAP part of the system allowed a flexible reporting at reasonable response times but added complexity and extra system management overhead.
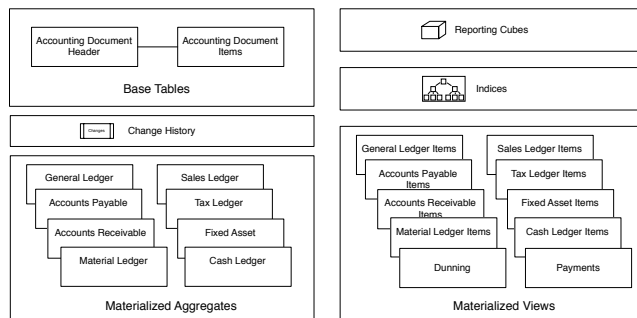

Figure 3: Schema of Financial System (Status Quo)

3.2 Customer Data Analysis

After analyzing the change logs of 4 different SAP customers we found that updates can be categorized into 3 major types:

- *Aggregate update:* The attributes are accumulated values as part of materialized views (between 1 and 5 for each accounting line item)

- *Status update:* Binary change of a status variable, typically with timestamps

- *Value update:* The value of an attribute changes by replacement

3.2.1 Aggregate Updates

Most of the updates taking place in financial applications apply to total records following the structure of the coding block. The coding block can contain e.g. account number, legal organization, year, etc. These total records are basically materialized views on the journal entries in order to facilitate fast response times when aggregations are requested. Since the roll-ups into multi-dimensional cubes became obsolete when data warehouses based on column storage were introduced [18] (see for example SAP Business Warehouse Explorer), we analyzed whether aggregates could be created via algorithms and always on the fly. The more instances of aggregates are requested, the better for the relative performance of the column storage (Figure 4). The creation of aggregates corresponds to a full column scan, therefore the number of aggregates in the response set has only little impact on the response time. In a row store, the response time increases linearly with the number of aggregates read.
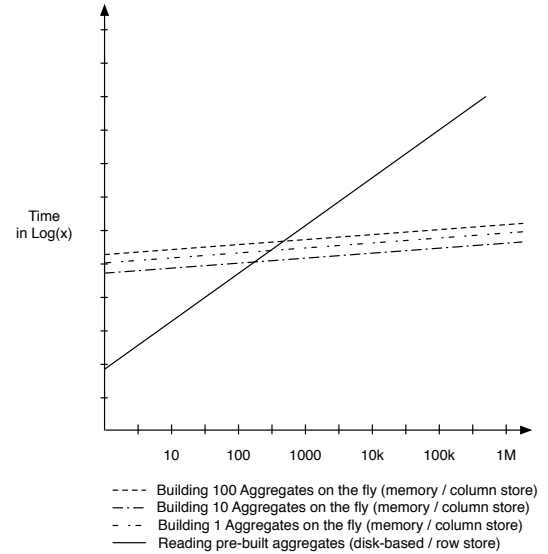

**Figure 4: Aggregation On the Fly vs. Read of Materialized Views**

3.2.2 Status Updates

Status variables (e.g. unpaid, paid) typically use a predefined set of values and thus create no problem when performing an in-place update since the cardinality of the variable does not change. It is advisable that compression of sequences in the columns is not allowed for status fields. If the automatic recording of status changes is preferable for the application, we can also use the insert-only approach, which will be discussed in Section 3.2.3, for these changes. In case the status variable has only 2 values, a null value and a timestamp are the best option. An in-place update is fully transparent even considering time-based queries.

3.2.3 Value Updates

Since the change of an attribute in an enterprise application in most cases has to be recorded (log of changes), an insert-only approach seems to be the appropriate answer. Figure 5 shows that on average only 5% of the tuples of

a financial system are actually changed over a long period of time. The extra load for the delta manager (the write-optimized store in a column store database which handles updates and inserts) and the extra consumption of main memory are acceptable. We only insert into the delta storage where the dictionaries are not sorted but maintained in the order of insertion. With insert-only, we also capture the change history including time and origin of the change.
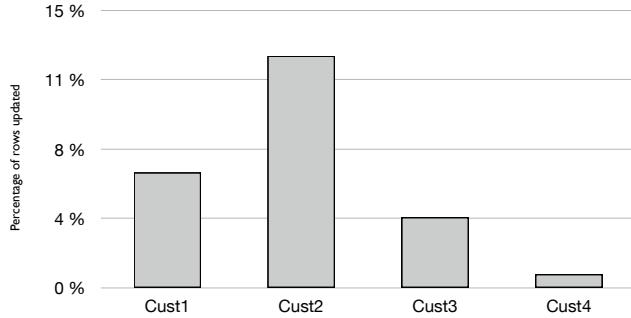
Figure 5: Financial Accounting Update Frequencies

Despite the fact that typical enterprise systems are not really update-intensive, by using insert-only and by not maintaining totals, we can even reduce these updates. Since there are less updates, there are less locking issues and the tables can be more easily distributed (partitioned) horizontally across separate computing units (blades) with a shared nothing approach [19]. Having basically eliminated updates, we now need to consider only inserts and reads. How we distinguish between the latest representation of a tuple and the older versions and maintain concurrency between read and update will be discussed in the next section.

With these recommended changes to the financial system, the number of major tables will drop from more than 18 (not including change history, indices, and OLAP cubes) to 2, as depicted in Figure 6. We only keep the accounting documents – header and line items – in tables. The insert-only approach and calculation algorithms executed on the fly replace all indices, materialized views, and change history.
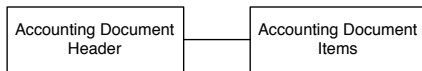
Figure 6: Simplified Financials System (Target)

4. CONSEQUENCES OF THE INSERT-ONLY APPROACH

The insert-only approach has consequences on how locking is handled both on the application- and database level.

4.1 Application-Level Locks

Many business transactions deal with several relational tables and multiple tuples of one table simultaneously. The applications "think" in objects, a layer which is established on top of the relational model. Although we can handle most concurrency situations without exclusive database locks we have to introduce a shared lock on object (i.e. application) level. The application object *customer*, for example, translates into up to 15 database tables. A change of a customer

object can include banking information, shipping address, etc. It is mandatory for consistency to have only one transaction changing this specific object at a time. Another example is the reconciliation of open items in accounts payable or receivable. Multiple open items will be marked as paid in one transaction. The lock is not on the accounting line items table but on the objects *creditor* or *debtor*. These application-level locks are implemented using an in-memory data structure.

4.2 Database Locks

With the insert-only approach the update of tuples by the application could be eliminated with the exception of binary status variables. Having multiple versions of the same tuple in the database requires that the older ones be marked as no longer valid. Each inserted tuple carries the timestamp of its creation and in case it is being updated, the timestamp of the update. Only the latest version of a tuple carries no update timestamp and is therefore easily identifiable. The benefit of this concept is that any state of the tuple can be recreated by using the two timestamps with regards to a base date for the query. This approach has been adopted before in POSTGRES [21] in 1987 and was called "time-travel". The extended SQL has to support a base date parameter through which the valid version of a tuple can be identified.

To carry all older versions of a tuple in the same table has significant application advantages especially in planning applications, where retrieving older versions of data is common [7]. In addition to that it completely eliminates the necessity of creating a separate log of the changes. The additional storage capacity requirements can be ignored.

The timestamp attributes are not participating in any compression algorithm and therefore do not lead to reorganization of the column when updated. Since multiple queries can coincide with inserts and updates, extreme care has to be taken to avoid too much locking on table-, column- or dictionary level.

Now we look at inserts. Inserts are added to the delta store at the appropriate partition of a table. The timestamp at the start of a query defines which tuples are valid (only tuples with a lower timestamp). In case an insert is in progress (single or multiple ones) the timestamp of the start of a new query will be set to the timestamp of the insert transaction minus one, and again the ongoing insert(s) will be ignored. This procedure is equivalent to snapshot isolation via timestamps [15, 3].

In the current research system we observed a significant increase in time per insert, when multiple queries were concurrently running. We believe this is an implementation issue from the days when maximum compression for read only applications was the design objective. Since the inserts are realized as an append to the delta store no exclusive lock should be necessary. If a transaction includes multiple inserts, the same logic as for the insert/update will apply. All queries running in concurrency will see a consistent snapshot via the comparison of timestamps.

Future research will specifically focus on concurrency and locking issues. As a general rule the data base system should perform each task with maximum speed, even occupying all resources (e.g. CPU cores) in order to reduce the potential for collisions and increasing management overhead.

Like the original inserts, all following inserts of changed tuples carry a globally unique user identifier. Together with

the timestamps this provides a complete change history.

Having the complete history of a tuple in the table allows the application to develop presentations of the evolution of facts over time. An example is the evolution of the sales forecast per day over a quarter in correlation with external events to better understand trends and improve the extrapolation (Figure 7). Despite the application induces a full table scan for each incremental move of the slider (see dashed line), the user experience is similar to using a scroll-bar in Microsoft Word.
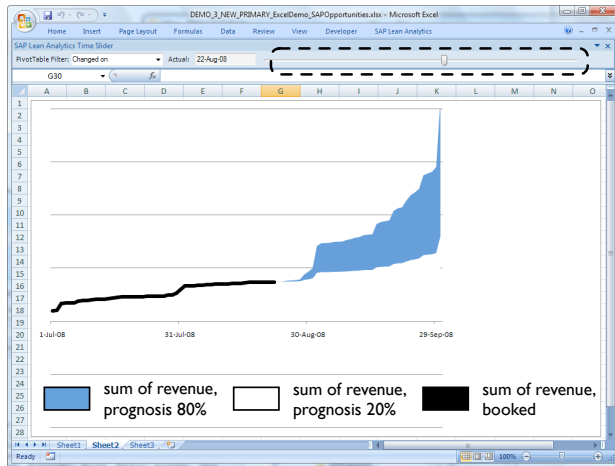
Figure 7: Sales Pipeline Forecast Using Historical Versions of the Data

5. COLUMN STORAGE IS SUPERIOR TO ROW STORAGE WITH REGARDS TO MEMORY CONSUMPTION

Under the assumption to build a combined system for OLTP and OLAP data has to be organized for set processing, fast inserts, maximum (read) concurrency and low impact of reorganization. This imposes limits on the degree of compression for both row and column storage. While it is possible to achieve the same degree of compression in a row store as in a column store (see for e.g. IBM's Blink engine [17]), a comparison of the two should be done assuming that the requirements above (especially fast inserts) are met, which excludes read-only row stores from the discussion.

In the column store, the compression via conversion of attribute values and the complete elimination of columns with null values only is very efficient but can be improved in this research system by interpreting the values: all characters blank, all characters zero, and decimal floating point zero as null values. Applications think in default values and do not handle null values properly. By translating the default values automatically into null values on the way into the database and back into default values on the way out.

Comparing the memory requirements of column and row storage of a table, the difference in compression rate is obvious. Various analyses of existing customer data show a typical compression rate of 20 for column store and a compression rate of 2 for (write-optimized) row storage on disk. For further memory consumption estimates we use a factor of 10 based on compression in favor of column storage. As

discussed in the previous sections, column storage allows us to eliminate all materialized views (aggregates) and calculate them algorithmically on demand. The storage requirements associated with these aggregates vary from application to application. The multi-dimensional cubes typically used in OLAP systems for materialized roll-ups grow with the cardinality of the individual dimensions. Therefore a factor 2 in favor of column storage based on the elimination of redundant aggregates is a conservative estimate.

Horizontal partitioning of tables will be used based on time and tenants. The option to partition into multiple dimensions is very helpful in order to use different qualities of main memory and processor speed for specific dimensions. Within the context of memory consumption the option to split tables into current data and historic data per year is extremely interesting. The analysis of customer data showed that typically 5-10 years of historic data (no changes allowed) are kept in the operational database.

Historic data can be kept accessible but reside on a much cheaper and slower storage medium (flash memory or disk). The current data plus the last completed year should be kept in DRAM memory on blades for the typical year over year comparison in enterprise systems. For the separation by time we use two time stamps, creation time and completion time. The completion time is controlled by the application logic e.g. an order is completely processed or an invoice paid. The completion date determines the year in which data can become historic, that means no further changes are possible. With regards to main memory requirements we can take a factor 5 in favor of column storage into account. It is only fair to mention a horizontal partitioning could also be achieved with row storage. Should the remaining table size for the current and last years partition still be substantial, horizontal partitioning by the database management may occur. Ignoring memory requirements for indices and dimension dictionaries, we can assume a $10 \times 2 \times 5$ time reduction in storage capacity (from disk to main memory). Next generation boards for blade servers will most certainly provide roughly 500 GB of main memory with a tendency of further growth. Since arrays of 100 blades are already commercially available, installations with up to 50 TB for OLTP and OLAP could be converted to an in-memory only system on DRAM. This covers the majority of SAP's Business Suite customers as far as storage capacity is concerned.

6. WHAT HAPPENS TO TYPICAL DATA-ENTRY TRANSACTIONS?

Data entry transactions consist of three parts: user data entry, data validation, and database update. Most of the data validation remains unchanged. Only the fact that any attribute of a table operates as an index can help to improve the quality of validation, e.g. in checking for duplicates of customer-, supplier-, parts-entries or incoming invoices. The database update is reduced to a mere insert. No indices (primary and secondary ones) need to be maintained and for journal entries, such as customer orders, stock movements etc., no update of aggregates takes place. As a result, the throughput of transactional data entry will improve. The delta manager handles the initial insert of new tuples.

The delta storage is again organized as a column storage. Since data retrieval and inserts can influence each other, extreme care has to be taken in the implementation to avoid

unnecessary locking. This is particularly true with inserts in partitioned tables. In order to reduce the influence of inserts on dictionary tables and reduce the impact of merge operations between delta storage and main storage a two tier organization of the delta storage is a concept currently investigated. The focus of research and development shifts consequently from maximum compression of data to high speed inserts with minimum effect on other simultaneously running queries.

7. THE IMPACT ON APPLICATION DEVELOPMENT

Applications based on a relational database using column storage should use the relational algebra and the extended SQL-features to delegate as much of the logic to the database level and the stored procedures. In rewriting existing applications we expect to reduce the amount of code by more than 30% (in more formal applications like financials 40-50%). Many parts can be completely restructured using the fully-indexed nature of column storage. In an ideal situation the application sets only the parameters for an algorithm which is completely defined by extended SQL (as a stored procedure) and executed on database level. The application then works on the result set to produce the output (screen, e-mail, print, phone, etc.). As mentioned before, the strict use of minimal projections is recommended. The high performance of the database makes caching of data on the application level largely superfluous.

The option to partition tables in multiple dimensions (tenant, time, primary key range, etc.) helps to achieve minimum response times for even larger tables. Since columns which have not yet been populated do not take any space in storage, except a 100 bytes stub, the addition of new columns to an existing table is simple.

To verify our findings, a research team started to set up a next generation accounting system for accounts receivable, accounts payable, general ledger and cost accounting including planning. The basis is SAP's on demand system ByDesign. All user interaction, configuration etc. remain identical to enable a complete parallel test.

The table for the journal entries has only one index, the accounting document number (plus line item number). No indices to connect the journal entries with the accounts (debtor, creditor, G/L or cost center etc.) exist. The only attributes updated in place are: creation-, invalidation- and reconciliation timestamp. All other changes result in an insert of the changed entry and the invalidation of the old one.

There are no aggregates in form of materialized views; they will instead be created via algorithms on the fly. The data entry speed improves since only two tables (document header, document line item alias journal entry) receive inserts. The simplicity of the transaction allows reconsidering a forward recovery approach instead of backing out a failed transaction.

Every presentation of accounting data can be defined as a spreadsheet, identifying the account(s), their hierarchical structuring (sets), the values to be calculated (aggregates). After a translation into extended SQL, the statement can be verified for correctness and, assuming the SQL processor works flawlessly, no further testing is required. The application can fully concentrate on user interaction and information presentation. In a second phase a research project including business experts will focus on the potential of a fully indexed data base with close to zero response time.

Not only have redundant tables been eliminated, but also their maintenance in form of update procedures or the ETL process between the OLTP and OLAP parts of a system.

8. COLUMN STORAGE IN SAAS APPLICATIONS

In SaaS (Software as a Service) applications several aspects of column storage are helpful. Columns which are unused are only represented by a stub. The introduction of a new attribute to a table means an update of the metadata and the creation of a stub for the column [2]. The attributes can from then on be used by the application. This is an important feature for the ongoing development of the application without any interruption for the user. The join with external data, which after import into the host system is held in column storage, is extremely efficient even for very large tables (minimum main memory accessed). In both cases the greatly improved response time will be appreciated.

Not only can the application now determine what base date for a query should be chosen but the development of the content (attributes) of individual tuples can be monitored (e.g. lifecycle of a customer order, control of sensitive data in human resources or accounts payable).

9. FUTURE RESEARCH

Our ongoing research efforts are concentrated on creating a benchmark for combined OLTP and OLAP systems, which is derived from real customer systems and data [4], multi-tenancy for in-memory column databases as well as optimizations around the delta merge process.

In future work we will investigate the following directions:

- Disaster recovery

- TCO (Total Cost of Ownership) comparison between the current version of SAP's on-demand accounting system and a version based on column store, including energy consumption.

- Extension of the model for unstructured data

- Lifecycle-based data management - Based on the semantics of different applications it is possible to specify if a single record is ever modified again or remains read-only and thus allows different strategies for compression and partitioning.

- Vertical Partitioning - In enterprise applications several chunks of a single relation tend to be grouped together by their access patterns. Using a vertical partitioning approach allows performance improvements when reading the content of those groups.

10. CONCLUSION AND OUTLOOK

The experience gained during the last 2.5 years encourages us to predict enterprise systems for even larger companies (e.g. up to 100 million sales activities per year), where all business transactions, queries, including unrestricted aggregations and time-based sequences, can be answered in just a couple of seconds (including the surprisingly costly presentation layer). We expect that the impact on management of

companies will be huge, probably like the impact of Internet search engines on all of us. Figure 8 shows a future management meeting with information finally at your fingertips [10] without any restriction.
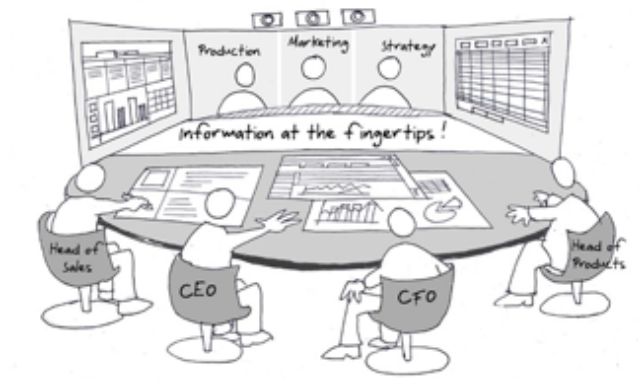
Figure 8: Management Meeting of the Future

Acknowledgements

11. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682. ACM, 2006.

[2] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-Tenant Databases for Software as s Service: Schema-Mapping Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1195–1206. ACM, 2008.

[3] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A Critique of ANSI SQL Isolation Levels. pages 1–10, 1995.

[4] A. Bog, J. Krueger, and J. Schaffner. A Composite Benchmark for Online Transaction Processing and Operational Reporting. In *IEEE Symposium on Advanced Management of Information for Globalized Enterprises*, 2008.

[5] P. Boncz. Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications. 2002. PhD Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65. Morgan Kaufmann, 1999.

[7] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[8] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 268–279. ACM Press, 1985.

[9] C. D. French. "One Size Fits All" Database Architectures Do Not Work for DDS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 449–450. ACM Press, 1995.

[10] B. Gates. Information At Your Fingertips. Keynote address, Fall/COMDEX, Las Vegas, Nevada, November 1994.

[11] J. Gray. Tape is Dead. Disk is Tape. Flash is Disk, RAM Locality is King. Storage Guru Gong Show, Redmon, WA, 2006.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[13] W. H. Inmon. *Building the Data Warehouse, 3rd Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[14] G. Koch. Discovering Multi-Core: Extending the Benefits of Moore's Law. *Technology@Intel*, (7), 2005.

[15] D. Majumdar. A Quick Survey of MultiVersion Concurrency Algorithms, 2007. http://simpledbm.googlecode.com/files/mvcc-survey-1.0.pdf.

[16] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), 1965.

[17] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 60–69. IEEE, 2008.

[18] J. Schaffner, A. Bog, J. Krüger, and A. Zeier. A Hybrid Row-Column OLTP Database Architecture for Operational Reporting. In *Proceedings of the Second International Workshop on Business Intelligence for the Real-Time Enterprise, BIRTE 2008, in conjunction with VLDB'08, August 24, 2008, Auckland, New Zealand*, 2008.

[19] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[20] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564. ACM, 2005.

[21] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.