



Using SQL-MapReduce[®] for Advanced Analytical Queries

**A Technical Whitepaper
Second Edition**

**Author:
Rick F. van der Lans
Independent Business Intelligence Analyst
R20/Consultancy**

September 20, 2011

Sponsored by Teradata

Table of Contents

1	Summary	1
2	From Simple Reporting to Advanced Analytics	3
2.1	Operational Analytics	3
2.2	Deep Analytics	4
2.3	Big Data Analytics	4
2.4	Semi-Structured Data Analytics	4
2.5	Self-Service Analytics	4
2.6	Complex Analytics	5
3	Advanced Analytics Requires Powerful Analytic Platforms	5
4	Using SQL for Advanced Analytics	7
4.1	Is SQL Right for Analytics?	7
4.2	Declarativeness and Storage Independency	9
4.3	Advantages of Declarativeness and Storage Independency	10
4.4	Why Are SQL Queries Sometimes Slow?	11
5	Parallelization of SQL	11
5.1	A Parallel Database Server	11
5.2	Different Forms of Query Parallelization	12
5.3	Parallelizing SQL Queries	13
6	Advanced Analytics Requires a Different Solution	14
7	Option 1: Do Nothing and Wait for Hardware Improvements	14
8	Option 2: Use a Different Language	15
9	Option 3: Extending SQL with (User-Defined) Functions	15
9.1	Built-In or User-Defined Functions	16
9.2	Scalar or Table Functions	16
9.3	Pure SQL, Procedural, or External Functions	16
9.4	Simple or Complex Functions	17
9.5	Parallelizing Function Processing	17
10	Option 4: Extending SQL with MapReduce	18
10.1	Introduction of MapReduce	18
10.2	Aster Database's SQL-MapReduce	20
10.3	Built-In Functions	21
11	Option 5: Use Apache's Hadoop	21
11.1	Architecture of Hadoop	21
11.2	Characteristics of Hadoop	22
11.3	Comparison of Aster Database and Hadoop	22
11.4	Summary	23
12	Technical Advantages of SQL-MapReduce	24
13	Business Advantages of SQL-MapReduce	25
14	Case Studies	26

15	Conclusion	27
	About the Author Rick F. van der Lans	28
	About Aster Data	28
	About Teradata	28
	Appendix A – The Built-in Functions of Aster Database	29

1 Summary

This whitepaper describes the advantages of merging SQL with MapReduce to create an analytic platform that can support today's complex and data-intensive query workload called advanced analytics. It focuses on the SQL-MapReduce[®] implementation offered by Teradata called Aster Database.

From the early days of decision support systems to the present world of data warehousing, business intelligence, and analytics, the need for analytical capabilities has increased. In the beginning, users were satisfied with relatively straightforward reports, while today they demand advanced online analytical capabilities, such as forecasting, predictive modeling, and clustering. In addition, data latency has become a more critical issue. It used to be acceptable for users to access week- and sometimes month-old data, compared to today, when users need to analyze data no older than a few seconds. In the past, we could keep our data warehouse relatively small by storing only aggregated data. Today, however, decision makers want to access everything from top-level aggregations to the most granular, detailed data. Users are also demanding more ad hoc, interactive, and exploratory reporting and analytical capabilities. This will unmistakably increase the frequency of analysis. In short, analytical needs have changed and are still changing, and this always means more complex analytical queries on more data. In this whitepaper, we call this new form of workload *advanced analytics*.

More complex queries and more data require more powerful analytic platforms. Through the years, various database vendors have researched, experimented with, and developed different solutions. Some have implemented data warehouse appliances built on dedicated and proprietary hardware, others have opted for developing in-memory solutions, some have focused on exploiting the processor cache, and still others have switched from storing data in a record-oriented style to a column-oriented style. As expected, some have even tried to combine a few of these solutions.

SQL-MapReduce[®] is a framework specifically designed for advanced analytical queries which allows developers to write highly expressive functions in languages such as Java, C#, Python, C++, and R, which are automatically parallelized across the cluster for high performance. It's the result of combining the most popular database language, SQL, with a programming model created by Google called *MapReduce*. The goal of MapReduce is to distribute as much processing over as many processors as possible. This whitepaper describes the SQL-MapReduce implementation offered by Teradata's *Aster Database* (formerly called *Aster Data nCluster*), an analytical database server. Aster Database is part of the *Teradata Aster MapReduce Platform*.

On the outside, Aster Database looks like any other SQL database server. It supports standard SQL and all the common APIs, such as ODBC and JDBC, so it can be accessed by all the popular analytical and reporting tools. What's inside makes Aster Database special. The product was specifically designed for analytics and reporting. Its unique Applications-Within[™] architecture runs analytic application logic inside the database, leveraging its massively-parallel architecture and SQL-MapReduce to fully parallelize processing of advanced analytical queries. The first version of Aster Database was released in 2006, and the first

production deployment was in 2007. The current Aster Data version 4.6.1 was released earlier in 2011.

Some of the characteristics of SQL-MapReduce:

- MapReduce is implemented as a set of SQL table functions. These functions, although they might be highly sophisticated internally, look like the table functions SQL already supports.
- Report developers don't have to learn a new language or a new set of statements. Instead they only have to study the parameters of the MapReduce functions.
- Every available reporting and analytical tool that supports SQL can work with SQL-MapReduce.
- SQL-MapReduce, as seen by report developers, is still as *declarative* and *storage independent* as SQL itself. They don't have to concern themselves with technical details, such as indexes, partitions, and buffer management parameters.
- Beside the built-in MapReduce functions, developers can write their own analytical functions. They can select their preferred language to write these functions, including Java, C++, C#, Python, and R.
- The complexity of MapReduce functions can range from simple functions doing simple selections to functions containing complete forecasting and optimization algorithms.

There are five reasons why SQL-MapReduce as implemented in Aster Database can considerably improve the performance of complex analytical queries:

- By using SQL-MapReduce instead of classic SQL, most analytical queries are simpler to formulate (less code), and therefore easier to optimize for the database server. This ensures a better query performance.
- Most of the analytical processing of a query is pushed to the MapReduce functions. This function code is distributed over as many nodes as possible, which means most of the query processing is done in parallel and not by a central module. Even complex forms of analytics are parallelized.
- The MapReduce functions are coded in a classic programming language and therefore execute very fast, especially if compiled languages, such as Java and C++, are used. In addition, the performance is quite predictable.
- The MapReduce functions don't process the data in a set-oriented way, but by each data item independently or by a partition of data. This gives the developer full control over how the code is executed, allowing him to program the most efficient processing strategy. There is a whole group of analytical queries for which the item-by-item or partition-by-partition approach are the more efficient processing styles.

- SQL-MapReduce is a software-based solution and can exploit various 'standard' cluster-based platforms. If new versions of the standard hardware components are released, SQL-MapReduce will be able to benefit from them.

To summarize, extending a SQL database server with MapReduce creates an analytic platform that combines the expressive query power and productivity of SQL with the parallelizability of MapReduce. The combination has the potential to improve the performance of complex analytical queries running on large to extremely large datasets. SQL-MapReduce is a powerful foundation for operational analytics, deep analytics, big data analytics, semi-structured data analytics, self-service analytics, and complex analytics. Teradata's Aster Database is a mature and robust implementation of SQL-MapReduce and has proven itself to be suitable for advanced analytics.

Note: The first edition of this whitepaper was published in June 2010. Since then, much has changed: Aster Data was acquired by Teradata, which led to the renaming of products; new versions of Aster Database offering new functionality were released; and, technologies such as Apache's Hadoop and Hive, have started to attract more attention. Therefore, it was decided to publish this updated and extended edition, which includes a section in which Hadoop and Aster Database are compared.

2 From Simple Reporting to Advanced Analytics

The reporting needs of managers and decision makers have changed over time. In the beginning, users were satisfied when they could run simple reports, such as "Get the total amount of sales per region." The list of reports they could run was normally pre-defined; they couldn't develop new reports themselves. This was done by reporting specialists in the IT department.

It didn't take very long before users requested more advanced analytical and ad hoc capabilities that would allow them to create new reports themselves. To address user needs, vendors released so-called *managed query tools* followed by *OLAP tools* (OnLine Analytical Processing). Both classes of tools gave users more query capabilities, such as drill-downs, rollups, and the ability to create new reports themselves. They were able to do all this without having to understand SQL or database technology. But the need for more analytical capabilities didn't stop there. Users continued to increase their demands and needs. Nowadays, users want to run highly complex statistical models that show, for example, the likelihood that certain customers will switch to a competitor.

Reporting and analytical needs will continue to evolve. Below are some of today's new forms of analytics. Each of these will undoubtedly have a serious impact on how analytic platforms will be used and on how they should be developed.

Operational Analytics – *Operational analytics* refers to a form of analytics applied by operational management. In most cases, the analytical needs of operational management require access to almost 100% up-to-date data, or in other words, (near) real-time data. Many examples exist where there is a need for operational analytics. For example, a retail company might want to know whether a truck already on the road to deliver goods to a specific store,

should be redirected to another store with a sudden, more urgent, need for those products. It would not make sense to execute this analysis with yesterday's data. Another example is credit card fraud detection. A classic form of credit card fraud is when stolen card data is used to purchase products. Each new purchase has to be analyzed to see if it fits the buying pattern of the card owner and whether the purchase makes sense. One of the checks could be whether two purchases in different cities occurred within a limited time of each other. For example, if a new purchase is made in Boston and the previous one was in San Francisco just a few seconds earlier, chances are high that this is fraud. But this form of analysis only makes sense when executed on operational data. The challenge, though, is that most data warehouses offer refresh rates of once-a-day or once-a-week.

Deep Analytics – For many reports and forms of analytics, storing detailed data is not necessary; aggregated data or slightly aggregated data is sufficient. For example, to determine the total sales per region, there is no need to store and analyze all the individual sales records. Aggregating the data on, for example, customer number is probably adequate. But for some forms of analytics, detailed data is needed. This is called *deep analytics*. If an organization wants to analyze whether trucks should be rerouted, or if it wants to determine which online ad to present, detailed data must be analyzed. And the most well-known area that requires detailed data is time-series analytics. But detailed data means the data store holding that data is going to grow enormously, potentially leading to serious problems with query performance.

Big Data Analytics – Many traditional information systems exist that store and manage large numbers of records. The last years, new applications have been built that store an amount of data magnitudes larger than those in these more traditional applications. For example, click-stream applications, sensor-based applications, and image processing applications, they all generate massive numbers of records per day. The amount of records here is not measured in the millions, but sometimes in the trillions. Analyzing this amount of data is a challenge. The new term introduced for this form of analytics is *big data analytics*.

Semi-Structured Data Analytics – Most data stored in SQL database servers can be classified as structured data, such as simple numbers, names, and dates. But more and more often the need exists to analyze data that can't be classified as structured, such as URL's, text documents, and audio streams. Evidently some structure in that data does exist, but that structure will be in the data itself, not in the data structure. For this type of data the term semi-structured is used, and the form of analysis to analyze this data is called *semi-structured data analytics*. The amount of semi-structured data organizations are collecting is increasing dramatically. For example, think about weblogs. If not already, analytics of large sets of semi-structured will be on everyone's agenda in the near future.

Self-Service Analytics – Before users can invoke their reports, the IT department has to set up a whole environment. *Self-service analytics* allow users to develop their own reports without serious help from the IT department. Self-service analytics is useful when a report has to be developed quickly and there is no time to prepare a complete environment. For example, an airline wants to know how many passengers will be affected by a particular strike tomorrow. And when a requested report will be used only once, self-service analytics can be helpful. In both cases, it would not make sense to first develop a data store that contains aggregated data before running the reports. For the first example, creating the derived data store would take too long, and for the second example, it's not worth the effort. Self-service analytics might also be called *unplanned analytics*, because the data warehouse administrators have no idea which

queries will be executed and when. This means it's impossible to optimize and tune the data warehouse for these queries.

Complex Analytics – The complexity of user demands keeps increasing. Besides standard reports, users want to create and run complex statistical models. They may want to create forecasting models (i.e. a retailer might want to see the impact of a price increase on expected sales), predictive models (i.e. an insurance company might want to predict which customers will be more interested in particular insurance combinations), and optimization models (i.e. a transportation company wants to know what the most efficient route is for a truck to deliver goods to various stores). Some of these models are pure data mining algorithms and require complex to very complex queries. And for some of them, large portions of the data warehouse must be scanned.

To summarize, all these new forms of analytics imply, on one hand the need for storing massive amounts of data for reporting and analytics is growing, and, on the other, the need for more complex forms of analytics is increasing. In this whitepaper we call this *advanced analytics*; complex and dynamic forms of analytics on massive amounts of detailed data; see Figure 1.

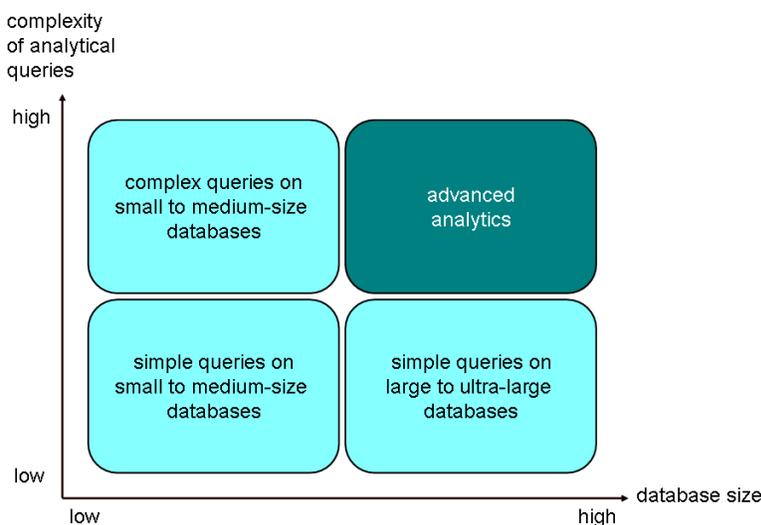


Figure 1 *The positioning of advanced analytics*

3 Advanced Analytics Requires Powerful Analytic Platforms

The big question is if the current data base servers can handle this workload required for advanced analytics? A study performed by TDWI¹ in 2009 shows that almost half of the organizations are considering switching to another data warehouse platform; see Figure 2. *“There’s probably a new generation in your near future. TDWI survey data shows that almost half of respondents are planning a data warehouse platform replacement in 2009–2012. Many others anticipate keeping their current platforms, but updating them significantly.”*

¹ [P. Russom](#), *Next Generation Data Warehouse Platforms*, TDWI Best Practices Report, fourth quarter, 2009.

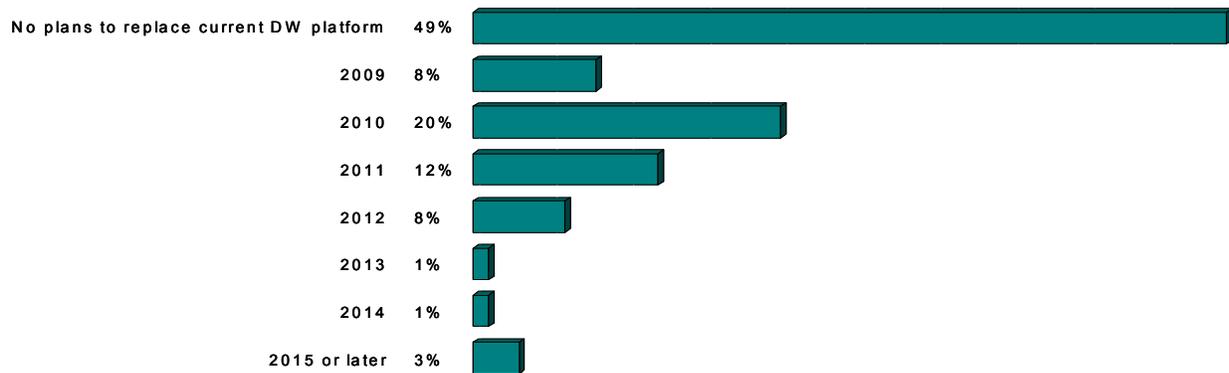


Figure 2 A TDWI study shows that close to 50% of the respondents are considering migrating to another data warehouse platform

But why do they want to switch platforms? The same study shows that quite a number of organizations is not satisfied with the performance of their current data warehouse platforms; see Figure 3. Imagine what would happen if in those environments even more data is added and the query workload is intensified. It would simply add to the existing problems.

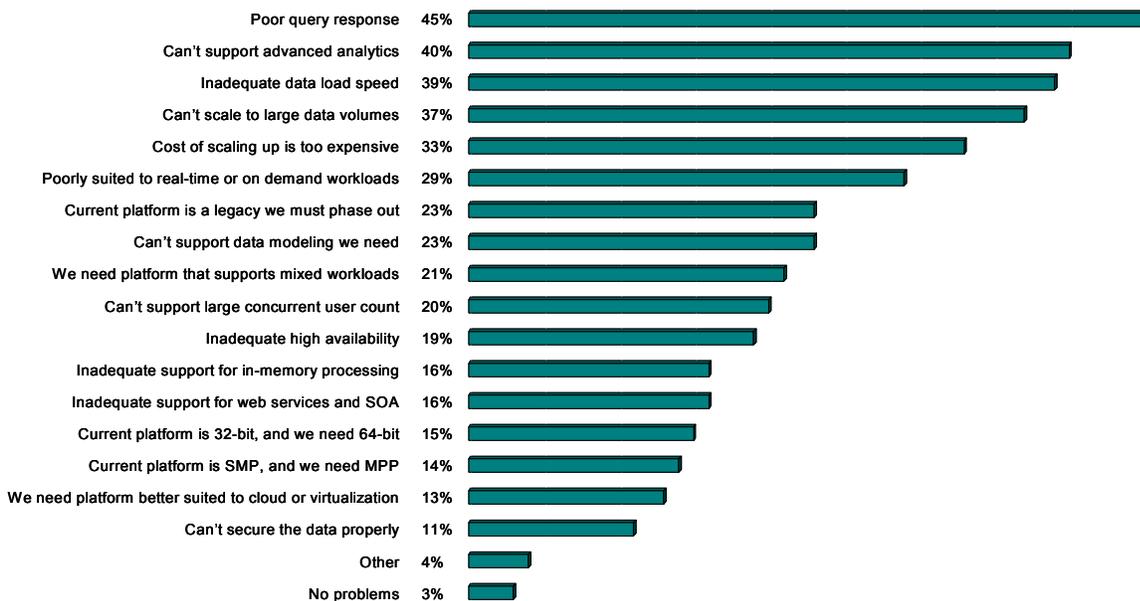


Figure 3 Reasons for switching to another data warehouse platform

Implementing advanced analytics will become a serious challenge. The need for more powerful platforms is real. But what does "a more powerful platform" really mean? More CPU power, more processors, more internal memory, and faster disks? Throwing more hardware at a problem doesn't always solve the problem, or at least not completely. For example, if a particular query has a performance of ten minutes and needs to be sped up with a factor of hundred, it won't help to use a machine that has processors twice the speed. The query will still be at least fifty times too slow. Sometimes what's needed is a software-based solution, one that

exploits hardware differently. This would imply a need for database servers that can exploit hardware more efficiently.

This begs the question, is SQL still the preferred language for implementing advanced analytics and are classic SQL database servers the preferred solution? If yes, is it necessary to extend the SQL language and should the SQL database servers be improved to make more efficient use of the hardware?

4 Using SQL for Advanced Analytics

Almost all database servers, young and old, support SQL. It's the most successful database language ever. It's the language implemented in the majority of available database servers, especially those specifically designed for analytics, sometimes called the *analytical database servers*.

Is SQL Right for Analytics? – SQL has always been a language with very strong query capabilities. In fact, in the 1970s and 1980s, SQL products were primarily employed for reporting and analytics. They did support transactions, but in this respect they were not as strong as the so-called hierarchical and network database servers, such as IMS, IDMS, and UDS.

Since the 1980s, the query capabilities of SQL database servers have improved and extended even further. Nowadays, SQL is able to support the most complex forms of reporting and analytics. It's hard to come up with a question that is impossible to formulate with SQL. The main challenge is, will the database server be able to run all those queries with acceptable performance? Let's illustrate this with a simple example.

Example: The following DEPARTURES table stores the scheduled departures of flights from a specific airport:

DEP_ID	DEP_DAY	DEP_TIME	DESTINATION	AIRLINE	DURATION
1	2010-04-01	14:20	London	Delta	9:30
2	2010-04-01	14:25	New York	Southwest	4:00
3	2010-04-01	14:50	New York	American Airlines	4:15
4	2010-04-01	15:10	London	American Airlines	8:50
:	:	:	:	:	:
16	2010-04-01	20:05	Paris	Delta	8:30
17	2010-04-01	20:15	Paris	Air France	8:40
18	2010-04-01	20:20	London	Virgin	9:00
19	2010-04-01	20:20	New York	American Airlines	4:00
20	2010-04-01	20:40	San Francisco	Southwest	3:30
21	2010-04-01	20:55	San Francisco	Delta	3:50
22	2010-04-01	21:00	New York	Delta	4:10
23	2010-04-01	21:35	London	British Airways	9:00
:	:	:	:	:	:

The user query: Get all the flights to London for which another flight exists to London that leaves within an hour on the same day:

```

SELECT  *
FROM    DEPARTURES AS D1
WHERE   DESTINATION = 'London'
AND     DEP_TIME + 60 MINUTES >=
        (SELECT  MIN(DEP_TIME)
         FROM    DEPARTURES AS D2
         WHERE   DESTINATION = 'London'
         AND     D2.DEP_TIME > D1.DEP_TIME
         AND     D2.DEP_DAY = D1.DEP_DAY)
ORDER  BY DEP_TIME

```

The result of this query returns a set of rows that includes row 1. However, for most database servers it's hard to process this query quickly, especially if the table contains millions of rows and if the DEPARTURES table has to be scanned several times. Additionally, if the table has been partitioned, it is questionable whether parallelization of the query will improve the performance.

Note this is a typical time-series type of query. The input data is selected according to the specified criteria and ordered by the specified timestamp column.

Of course this is a simple example, and in reality this DEPARTURES table will definitely not contain millions of rows. But it's easy to come up with comparable queries for which millions or even billions of rows have to be accessed. For example, in a credit card environment, we want to see whether two charges on a credit card didn't happen too close together in a period of time. Or, in a website environment, we want to determine how many different sessions were started by one user, where a session is defined as a number of clicks on the website with limited time in between.

The more complex the SQL query and the larger the data set, the bigger the chance the SQL database server is not able to come up with a fast processing strategy. Take the following question: Get all three items that are frequently purchased together in the same retail transaction by users. This question is like a market basket analysis. The corresponding SQL query is lengthy and is very hard to optimize for most database servers.

```

SELECT  A.PROD_DESC AS ITEM1, B.PROD_DESC AS ITEM2, C.PROD_DESC AS ITEM3,
        COUNT (*) AS CNT
FROM    (SELECT  SF.STORE_ID, SF.REG_ID, SF.TRAN_NO, SF.ITEM_ID, SF.DT, PD.PROD_DESC, PD.PRICE
         FROM    SALES_FACT SF, PRODUCT_DIM PD
         WHERE   SF.ITEM_ID = PD.ITEM_ID) AS TRANSACTIONS A,
        (SELECT  SF.STORE_ID, SF.REG_ID, SF.TRAN_NO, SF.ITEM_ID, SF.DT, PD.PROD_DESC, PD.PRICE
         FROM    SALES_FACT SF, PRODUCT_DIM PD
         WHERE   SF.ITEM_ID = PD.ITEM_ID) AS TRANSACTIONS B,
        (SELECT  SF.STORE_ID, SF.REG_ID, SF.TRAN_NO, SF.ITEM_ID, SF.DT, PD.PROD_DESC, PD.PRICE
         FROM    SALES_FACT SF, PRODUCT_DIM PD
         WHERE   SF.ITEM_ID = PD.ITEM_ID) AS TRANSACTIONS C
WHERE   A.STORE_ID = B.STORE_ID
AND     B.STORE_ID = C.STORE_ID
AND     A.STORE_ID = C.STORE_ID
AND     A.REG_ID = B.REG_ID
AND     B.REG_ID = C.REG_ID
AND     A.REG_ID = C.REG_ID
AND     A.TRAN_NO = B.TRAN_NO (continues on next page)

```

```

AND      B.TRAN_NO = C.TRAN_NO
AND      A.TRAN_NO = C.TRAN_NO
AND      A.DT = B.DT
AND      B.DT = C.DT
AND      A.DT = C.DT
AND      A.ITEM_ID <> B.ITEM_ID
AND      A.ITEM_ID <> C.ITEM_ID
AND      B.ITEM_ID <> C.ITEM_ID
GROUP BY A.PROD_DESC, B.PROD_DESC, C.PROD_DESC
HAVING   COUNT(*) > 1000
ORDER BY COUNT(*) DESC

```

To perform such a market basket analysis, the data warehouse has to keep track of what each individual customer buys and when. These tables normally contain millions and millions of rows. This means a highly complex query is executed on a very large database. It will be hard for most database servers to run this query quickly. In fact, sometimes these queries become so slow, that users are not allowed to run them online anymore, or worse, they are not allowed to run them at all. This would definitely limit the analytical capabilities.

Declarativeness and Storage Independency – Why are some of those queries so slow and why doesn't a database server always come up with a perfect processing strategy? Many factors exist that influence the performance of queries, but two fundamental properties of SQL, *declarativeness* and *storage independency*, have a big impact. Those two properties are and always have been fundamental to SQL. They were the basic design principles when the language was initially designed in the IBM labs^{2, 3, 4}.

When SQL was developed in the 1970s, it was supposed to be a *declarative* language. In this case, declarative means a SQL developer only has to program what has to be done, and not how it should be done. For example, in the next query we only specify that we're interested in customers headquartered in New York:

```

SELECT  *
FROM    CUSTOMERS
WHERE   LOCATION = 'New York'

```

Nowhere in this query do we specify anything that relates to how the query should be processed. For example, no loops are programmed. The database server itself has to determine how to get the requested data from the database to the user.

The second property of SQL is *storage independency*. This means the language should hide how data is physically stored and accessed. For example, when a query is specified, nowhere do we specify that a particular index should be used, nor do we specify the physical location of the table, we don't indicate that intermediate results should be kept in memory, or the order in which rows should be retrieved from disk. All these technical aspects are hidden for the SQL developer.

² R.F. Boyce, and D.D. Chamberlin, 'Using a Structured English Query Language as a Data Definition Facility', *IBM RJ 1318* (December 1973).

³ D.D. Chamberlin et al, 'SEQUEL 2: A unified approach to Data Definition, Manipulation and Control', *IBM R&D* (November 1976).

⁴ D.D. Chamberlin, 'A Summary of User Experience with the SQL Data Sublanguage', *IBM RJ 2767* (March 1980).

These two language properties can be regarded as independent of each other. For example, it's possible to design a language that is non-declarative but storage independent, and one that is declarative and storage dependent. Again, SQL is both declarative and storage independent.

Advantages of Declarativeness and Storage Independency – The main advantages of these two properties are improved productivity, maintainability, and flexibility:

- **Productivity:** Having to write declarative code and not having to deal with the 'how', implies having to write less code. This seriously improves the time needed to write code compared with having to write the equivalent solution in a non-declarative language. In addition, if a developer doesn't have to concern himself with details related to storage and access, it means less code again.
- **Maintainability:** For maintenance the same rules apply as for productivity: less code implies having to maintain less code. And the storage independence property means the maintenance programmer doesn't have to study the storage characteristics in order to make the change.
- **Flexibility:** Because SQL is storage independent, changes to the storage layer, such as table structures, indexes, and partitions, can be made without the need to change the SQL code.

These properties stem from the relational model, the theory on which SQL is based. The founder of the relational model, Edgar F. Codd, indicated in his seminal paper⁵, written in response to his receipt of the ACM Turing award, that his goal for developing the relational model was *data independence* (which in this whitepaper relates to storage independency): *"The most important motivation for the research work that resulted in the relational model was the objective of providing a sharp and clear boundary between the logical and physical aspects of database management [...]. We call this the data independence objective."* Also see Figure 4.

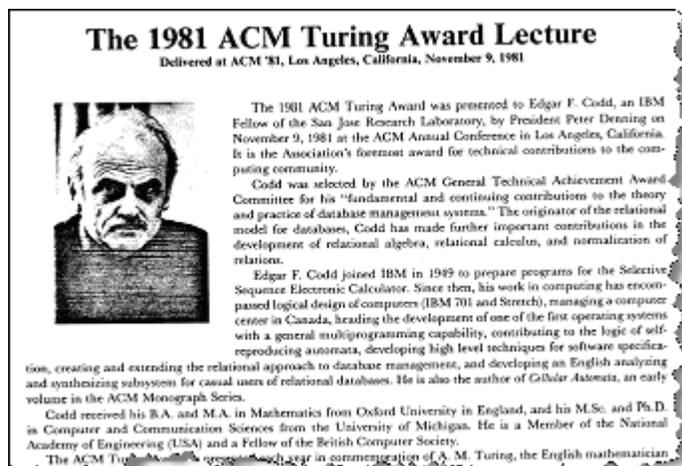


Figure 4 Edgar F. Codd's ACM Turing Award lecture

⁵ E. F. Codd, *Relational Database: A Practical Foundation for Productivity*, Turing Award Lecture in *Communications of the ACM* 25, no. 2 (February 1982).

Why Are SQL Queries Sometimes Slow? – Why can declarativeness and storage independency have a negative impact on performance? Because of these properties, a SQL database server has to transform a query into an *access plan* that describes in detail how the data should be accessed, joined, grouped, filtered, and so on. The access plan is not declarative and is not storage independent. It's a precise, step-by-step description of how data should be accessed. It contains references to indexes and specifications on how to parallelize the query.

It's the *optimizer*, a module belonging to the database server, that is responsible for transforming a query into an access plan. The smarter an optimizer is, the faster the queries. Through the years, the quality of optimizers has improved, but still some queries remain for which it is hard to come up with an efficient access plan. And that's one of the main reasons why performance is not always ideal.

To summarize, the declarativeness and storage independency of SQL are beneficial to productivity, maintenance, and flexibility. Even users with limited knowledge of databases can write queries. But the drawback is some queries will perform poorly because they are hard to optimize.

5 Parallelization of SQL

Most database servers on the market today can exploit multi-processor hardware when running queries. They distribute the processing of those queries over multiple processors. This is called *parallelization of queries*. This section explains why parallelizing queries can improve performance, and also that not all queries can be parallelized. We begin by introducing some terminology.

A parallel database server – In order to distribute query processing over multiple processors, the architecture of the database server is distributed. Figure 5 shows the typical architecture for a parallel database server. The database server has processing modules, frequently called *nodes*. One of those nodes is called the *Master* or the *Queen*, and the other nodes are *Workers*. Each Worker manages a number of tables or table partitions. Usually, the Master knows where all the data is stored. The Master and the Workers can run on different processors in one single machine, or they can be distributed over a network or cluster of machines.

When an application sends a query to the database server, it's first sent to the Master; that's where all the processing starts. Depending on which tables are accessed, which nodes those tables are located, and how the tables are partitioned, the Master will break a query in a number of smaller queries. Next, these so-called *query snippets* are distributed across the Workers. The Workers process the query snippets and return intermediate results back to the Master. The Master merges all those intermediate results into one final result, does some extra processing, if needed, and the final result is returned to the application.

In some database server architectures, a Worker can also play the role of Master. In other words, when such a Worker receives a query snippet, that snippet is again broken into even smaller snippets and those are sent to lower level Workers. These Workers send their results back to the Worker/Master. The latter combines all these results and returns the combined

result to the real Master. This process continues for every level of Workers. This type of architecture makes it possible to exploit clusters with high numbers of processors.

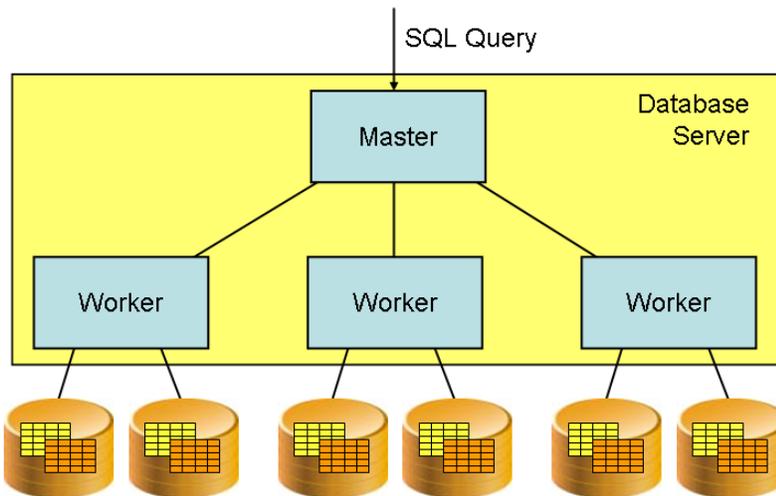


Figure 5 Typical architecture for parallel database servers

The main goal of this architecture is to let Workers do as much as possible in parallel, and to let the Master(s) do as little as possible, so that the Master does not become a potential performance bottleneck.

Different Forms of Query Parallelization – Different forms of query parallelization exist. *Inter-query parallelism* means the query workload is parallelized: different queries run on different nodes. Another form, called *intra-query parallelism*, is where the processing required for one particular query is distributed over multiple nodes. The first form improves the workload, whereas the second improves the response time of individual queries.

Of the second form, two sub-forms exist: *inter-operation parallelism* and *intra-operation parallelism*. A query must be broken into a set of *operations* for processing. An operation can be a sort, a scan, a join, or a projection. With inter-operation parallelism, the processing of different operations (belonging to one query) is distributed over multiple nodes. This can definitely improve the response time of a query. However, if one of the operations involves a scan of 500 million rows, and that scan is not parallelized, that one operation can still take minutes to complete, therefore slowing down the processing of the whole query.

With intra-operation parallelism the processing of an operation, such as a scan, is distributed over multiple nodes. Intra-operation parallelism does require partitioning of the table being scanned. If the table has been partitioned, and the partitions are assigned to different nodes and disks, they can be scanned simultaneously. This shortens the response times of the overall query. The big advantage is queries on extremely large tables can still be processed with fast response times. Intra-operation parallelism is especially relevant for complex analytical queries because of the need to process huge amounts of data.

Note this whole notion of parallelization is hidden for the SQL developer. Again, this is also part of the storage independency property of SQL. Developers don't and shouldn't have to indicate how to parallelize the query. This would make the queries too dependent on the current storage and partitioning structure of the tables.

Parallelizing SQL Queries – But how easy is it for a database server to parallelize queries, or how easy is it to offload processing from the Master to the Workers? In other words, what types of operations can be done in parallel by the Workers? Here are a few examples of how complex parallelization can become. Let's start with the following simple query:

```
SELECT  ID, SALES_DATE, PRICE
FROM    SALES_RECORDS
WHERE   PRICE > 100
```

For most products this query is easy to parallelize. The Master can send this whole query as a snippet to each Worker. And each of the Workers will only return those rows (and a few columns) for which the condition `PRICE > 100` is true. As indicated, this query is easy to parallelize, in fact the whole query can be parallelized. The only thing left for the Master to do is combine the results from the Workers using a simple union operation.

But what if the condition contains complex calculations, or a subquery, or if it invokes a complex user-defined function? Hopefully the database server is smart enough to include as much processing inside the query snippets, so the Workers send back the smallest possible results, and the Master only has to combine and sort these results and return them to the application. If this doesn't happen, and too many rows are retrieved from the disk and send back to the Master, the Master has to perform all the extra processing serially. This will, without a doubt, have negative impact on overall performance because too much query processing is not done in parallel.

Most analytical queries contain group-by operations, therefore the next example retrieves sales data per region and contains such an operation:

```
SELECT  REGION_ID, SUM(PRICE)
FROM    SALES_RECORDS
WHERE   PRICE > 100
GROUP BY REGION_ID
```

For many database servers it's hard or impossible to perform group-by operations in parallel. They would probably send the following query as snippet to the Workers:

```
SELECT  REGION_ID, PRICE
FROM    SALES_RECORDS
WHERE   PRICE > 100
```

This snippet doesn't contain any group-by operation. The consequence is many rows are returned to the Master, and the whole group-by operation is executed serially by the Master. Normally, a group-by operation groups sets of rows into individual rows. It's a lot more efficient if those group-by operations are performed by the Workers, because a much smaller set of rows would be returned.

As the next example we use the first query in Section 4. What should an optimizer do with the correlated subquery? What we don't want is the subquery to be executed for each row in each table because that might lead to an execution of the subquery for each individual row. Plus, it means for each row the subquery is sent to the Master to be processed. This will happen over and over again. In addition, if the table accessed in the subquery is large, it will be very slow. It's very hard for this query to come up with a fully parallelized access plan.

Time-series based queries are also hard to parallelize. In these queries, rows are selected based on a comparison between the row's values and the values of the previous or next row. For example, imagine we only want to select rows where the value of a column is greater than the value of the same column in the previous row. In most database servers, the Workers return all the rows to the Master and the row selection process is handled by the Master itself. This is inefficient.

To summarize, the challenge for every database server is to optimize the queries in such a way that as much processing is offloaded from the Master to the Workers. The more work that can be offloaded, the more processing is done by the Workers in parallel, and the better the performance. The complexity of advanced analytical queries makes it a major challenge for database servers to optimize them.

6 Advanced Analytics Requires a Different Solution

To come back to the question of whether SQL is suitable for advanced analytics, the problem is not that SQL isn't rich enough with respect to functionality (as described), the challenge is whether a database server is able to optimize the query in such a way that it can fully exploit parallel hardware to improve performance. In other words, how can these complex queries be executed efficiently on hundreds of millions of rows? Vendors can select from the following options which are described in the next sections:

1. Do nothing and wait for hardware improvements.
2. Use a different language, one that is easier to optimize and parallelize.
3. Extend SQL with (User-Defined) Functions.
4. Extend SQL with MapReduce.
5. Use Apache's Hadoop.

7 Option 1: Do Nothing and Wait for Hardware Improvements

From everyone's perspective, the preferred approach is that the database vendors do not change their SQL languages. Because no one would have to learn a new language or new extensions to the language. All reporting and analytical tools currently available would still be able to exploit the language. And the advantage to the vendors is that they wouldn't have to develop new features. However, it would mean the current query performance would still be slow.

This option would imply we have to wait for improvements on the hardware side to speed up queries. The disadvantage of this option is we won't know how long we have to wait. Is it one month, one year, or ten years? For some organizations this is not an acceptable option because they have urgent needs with respect to advanced analytics.

8 Option 2: Use a Different Language

Using a language different from SQL might be an option. It must be possible to invent a language specifically designed for analytics, one that has comparable or even more features than SQL and that can be parallelized more easily. Unfortunately, that language doesn't exist yet.

Potentially, this is an interesting option but there are some disadvantages. A major disadvantage is most tools for analytics and reporting won't support that language until their respective vendors have implemented support for it. The second disadvantage is all the developers have to learn a new language.

Although a specific language for *advanced* analytics doesn't exist yet, a language specifically designed for analytics does exist and has been on the market for quite some time now: Microsoft's MDX (MultiDimensional Expressions). MDX was first introduced as part of their OLE DB for OLAP specification in 1997 and has been widely adopted by the market. The specification was quickly followed by a commercial release of Microsoft OLAP Services 7.0 in 1998 and later by Microsoft Analysis Services.

Many vendors have currently implemented this language, including Oracle/Hyperion, MicroStrategy, SAS, and SAP, and vendors on the client side, such as IBM/Cognos, SAP/Business Objects, and Microsoft. However, all these implementations of MDX are definitely suited for analytics, but are not (yet) capable of handling very large datasets, which is a requirement for advanced analytics. In other words, MDX is not (yet) an alternative for advanced analytics.

9 Option 3: Extending SQL with (User-Defined) Functions

Another option to make SQL more fitting for advanced analytics is by adding new functions designed specifically for analytics. The concept of function is not new to SQL, in fact the first versions already supported functions, although those first functions were simple ones, such as truncate a string and calculate the square root.

When discussing functions, a distinction has to be made between two groups of developers. On one hand, there are developers who program the functions, and on the other, there are developers who write SQL statements that invoke the functions. To keep SQL's declarative and storage independent properties intact, it's important the SQL developers not be concerned with how functions work, in which language they have been coded, and so on. That should only be relevant to the developer of the functions.

Functions can be classified in many different ways. In this whitepaper, the following four classifications are used.

Built-in or User-defined Functions – The first classification is based on who the function developer is. Each database server comes with a set of functions developed by the vendor itself. These are called *built-in* or *standard functions*. The SQL developers have no idea in which language these functions are coded, how their internal algorithms work, and whether their processing can be parallelized, nor do they need to know.

In contrast, SQL also supports *user-defined functions* (UDFs). UDFs are coded by SQL developers themselves. This gives the developer full control over how the function is programmed.

Whether a function is built-in or user-defined, the SQL developer doesn't see the difference between those two types of functions; see the next example:

```
SELECT  FLIGHT, TRUNCATE(DEPARTURE_TIME, MINUTES)
FROM    DEPARTURES AS D1
WHERE   BANK_HOLIDAY(DEPARTURE_TIME) = 1
```

In this example, TRUNCATE is a built-in function, whilst BANK_HOLIDAY is a user-defined function that determines whether a specific day is a bank holiday. But the developer who writes the SQL statement doesn't see the difference between those two types of functions. For him the SQL code is still declarative. Although the developer who wrote the function BANK_HOLIDAY might have used a non-declarative language, such as Java and C++.

Scalar or Table Functions – The second way of classifying functions is by the result they return. There are functions that always return one scalar value, such as a string, a date, or a number, and there are those that return a set of rows where each row consists of the same number of values. The former group is usually referred to as *scalar functions*, and the latter group is called *table functions*.

The functions TRUNCATE and BANK_HOLIDAY are both examples of scalar functions. Other examples might be a function that changes a dollar value into a euro value, and one that subtracts an average value from a specific row value. Scalar functions can be used, for example, in the conditions of WHERE clauses to select rows, or in SELECT clauses to transform values of a row. An example of a table function is LAST_FIVE_ROWS, which returns the last five rows of a table (the ones with the highest primary key value). Another example could be a function that reads records from a sequential file stored outside the database and presents those records as rows. Table functions are mostly used in the FROM clause:

```
SELECT  AVG(DURATION)
FROM    LAST_FIVE_ROWS(DEPARTURES)
```

Pure SQL, Procedural, or External Functions – The third way of classifying functions is based on the language used for coding the function:

- Pure SQL functions: The bodies of these functions consist of one or two pure SQL statements. With pure SQL we mean the classic declarative SQL statements.

- **Procedural functions:** The bodies of these functions are written in declarative SQL statements and non-declarative statements, such as while-do and if-then-else. Those non-declarative statements are part of the SQL language itself and are processed by the database server. In Sybase these extra statements are called Transact-SQL, in DB2 they are called SQL PL, and in Oracle PL/SQL (where PL stands for Procedural Language). Most of these non-declarative statements resemble comparable Pascal and Ada statements, but are proprietary.
- **External functions:** The bodies of these functions are developed in 'external' languages, such as Java, C#, or possibly even Cobol. They might contain declarative SQL statements. The database server doesn't typically process these external functions, as an application server or a special engine is typically responsible.

Simple or Complex Functions – The fourth way of classifying functions is based on whether the body of the function contains queries itself. A *simple function* doesn't and a *complex function* does. For example, if a function contains only a calculation, it's a simple function. But a function that determines whether the value of an input parameter is less than the average value of a column, probably needs to query a table, which makes it a complex function.

Next let's focus on whether by moving logic into functions the performance of our analytical queries can and will improve, and whether the processing can be parallelized better.

Parallelizing Function Processing – Parallelizing simple scalar functions is not that difficult for a database server. It will push the processing of those functions down to the Workers. For example, if a function appears in a condition, such as `function(column) = value`, the processing of this condition can be moved to all the Workers, and each Worker only returns those rows to the Master that adhere to this condition. But this assumes all the data needed to evaluate the condition is available in the row being studied.

Parallelizing complex functions is more difficult. Imagine a scalar function called `NEXT_FLIGHT` has been developed that simplifies the long query in Section 4. It determines whether there is another departure to the same city leaving within one hour. This function has three input parameters and returns a 1 if there is another flight, and 0 if there isn't. A rewrite of the query but now with the function would look like this:

```
SELECT *
FROM   DEPARTURES AS D1
WHERE  DESTINATION = 'London'
AND    NEXT_FLIGHT('London', DEPARTURE_DAY, DEPARTURE_TIME) = 1
ORDER BY DEPARTURE_TIME
```

Obviously, due to this function, it's easier to write the SQL statement. But does it have a better performance? Probably not. Whatever way the function is written, it must use one or more extra queries to find another row. There is no other way to get to the rows than by using SQL statements, and this is regardless of the language in which the function is coded. If those extra queries are executed, they are sent to the Master which has to determine how to execute them. If this is done for each row, we will see an avalanche of queries being sent to the Master. Evidently, this will be very time consuming.

Some functions are deterministic. The property of a deterministic function is that if it's executed multiple times in a query it always returns the same value. When such a function is used in a query, the Master could execute it first and substitute the function call in the condition by its return value. The Master could then send the query with the substitution to the Workers. In this case the Workers wouldn't fire off extra queries and the function processing can easily be parallelized.

In the above SQL example, however, the function is not deterministic. For each row the function has to be executed, thus for each row, a query has to be executed again. So instead of processing one query, the database server has to process millions of small queries. This is not efficient. Note that it's not the procedural code that creates the problems, but the additional queries inside the function.

To summarize: It should be possible to extend SQL with functions that will simplify the formulation of complex SQL queries. And because the internal workings of the functions are not visible to SQL developers, SQL will stay declarative and storage independent. However, it remains difficult to parallelize the queries inside the functions. This is because SQL statements are needed again reason for extra data access.

Note: Extending SQL with functionality by adding functions has been done quite often during the lifetime of SQL. For example, functions have been added to support manipulating and querying XML documents. Some even have functions that makes it possible to extend SQL statements with XPath and XQuery statements. In this case, SQL operates as a host language for those other languages. Others have extended SQL by adding functions for data mining algorithms.

10 Option 4: Extending SQL with MapReduce

An option to make database servers more suitable for advanced analytics, proposed by some of the newer database vendors, is by extending SQL with *MapReduce*. Aster Database is one of those database servers. They have implemented the MapReduce programming model to exploit parallel hardware and to fully parallelize query processing and therefore make advanced analytics possible even on commodity hardware. Note that while other vendors have implemented this option, each implementation differs.

Introduction of MapReduce – Although MapReduce is much younger than SQL, it's used by more people than SQL will ever be. The reason is Google. If we search for a specific term with the Google search engine, we use their technology which is based on MapReduce. MapReduce is used for offline batch processing to build the search indexes. Then, when someone does a search with Google, the lookup is into those indexes. So even though we may be unaware of it, most of us use MapReduce daily. But what is MapReduce?

In 2004 two Google engineers published an article entitled *MapReduce: Simplified Data Processing on Large Clusters*⁶. In that paper they introduced MapReduce, a programming

⁶ [J. Dean and S. Ghemawat](#), 'MapReduce: Simplified Data Processing on Large Clusters', in Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, San Francisco, CA, December 06 - 08, 2004.

model for processing requests on large datasets where the processing can be distributed over a high number of nodes using parallel processing capabilities. Currently, MapReduce is also an implementation used by Google and it's been patented⁷ since January 2010. Google and other organizations use this programming model to maximize the processing parallelization and by doing that, improve response times.

We want to emphasize that MapReduce is a *programming model*, not a programming language. It's a style of solving a specific problem using a programming language. In principle, a programmer can use any programming language for implementing a MapReduce-based solution, Java, C#, Python, and so on.

The words *Map* and *Reduce* stand for the two steps in which a specific request is processed. During the Map step, a request is broken into smaller requests that are distributed over the Workers (for convenience sake, the terms Masters and Workers are used here as well). Possibly, those Workers break the request into even smaller ones that are distributed even further. In this case the Workers operate in a hierarchical manner, with multiple layers of Workers and Masters. So a module acts as Master and Worker.

A request in most cases is a function call (or procedure call or method invocation depending on the programming language). These map functions are coded by developers and can be as complex as they want. Calls to these functions are distributed over as many nodes as possible. Note there is a strong resemblance here with breaking a SQL query into multiple scan operations and distributing them over as many nodes as possible.

Let's illustrate this Map step with a simple example. Imagine we want to execute the function `GET_TOTAL_SALES_PER_STORE` on a dataset (which could be a simple file) that contains sales transactions. For each sales transaction, the customer id, the store id, the product id, the timestamp, and the product's sales price is stored. So for each individual product bought by a customer, a record is stored in this dataset. The dataset is partitioned over all the nodes. Also, imagine, the input parameter of this function is `MIN_AMOUNT`, which means, only those records should be included in the final result whose values are higher than the value of the parameter. The result of the function should be a set of records that shows the total amount of sales for each store. In the Map step, this call is distributed over as many nodes as possible.

During step 2, the Reduce step, all calls are executed (maybe hundreds in parallel) on different partitions of the dataset and the intermediate results are returned to the Master(s). This step is called Reduce, because only a reduced number of records is returned to the Master. As with parallel databases, the goal is to minimize the amount of data returned to a Master. What the logic of the reduce function looks like is up to the programmer. He can use any construct of the programming language to make this function as efficient as possible. Even the way the records are accessed in the dataset is determined by the reduce function. In other words, the code inside the functions is probably non-declarative and storage dependent, to make it as fast as possible.

⁷ [J. Dean et al.](#), 'System and Method for Efficient Large-scale Data Processing', United States Patent 7,650,331, January 19, 2010.

At the end of the reduce function, the Master combines all these intermediate results into the final result: a set of records.

Aster Database's SQL-MapReduce – Next, we describe how Teradata has integrated MapReduce with SQL to make Aster Database suitable for advanced analytics.

From the SQL developer's perspective, the whole framework is implemented as a set of *external table functions* that can be invoked from SQL. Some of these functions are built-in and developers can develop user-defined ones. So, a report developer won't have to learn a new language. He or she only has to learn what the parameters mean and how to invoke the new table functions. SQL will stay a declarative and storage independent language, and the MapReduce table functions are still inline with Ted Codd's ideas of the relational model.

Let's use an example to show what MapReduce means to these developers. We use the table plus the complex query from Section 4. If this query is rewritten using a MapReduce function, the query becomes straightforward:

```
SELECT *
FROM   GET_NEXT_FLIGHT_1HR (ON DEPARTURES PARTITION BY DESTINATION)
WHERE  DESTINATION = 'London'
ORDER BY DEPARTURE_TIME
```

Obviously, this query is a lot simpler to formulate than the original one. The FROM clause contains the call to the MapReduce table function GET_NEXT_FLIGHT_1HR. This function has two parameters. The first indicates the table to be queried (DEPARTURES), and the second specifies the column on which to group the rows (DESTINATION). The function returns a set of rows. It determines for each row in the DEPARTURES table whether there is a row with the same destination within one hour on the same day. Those rows form a group. The only thing the main query has to do is find the ones with destination London. Note that this function could have contained the condition as well, however, the function would no longer be usable for other columns, but only for columns containing city names.

Because of the MapReduce function, the query becomes easier to formulate, but more importantly, Aster Data can parallelize the query with the MapReduce functions much more easily than the original queries. The function contains the group-by operations plus the time-series piece (find another row) and both are fully parallelized.

This example doesn't really show the power of these MapReduce functions. Therefore, let's rewrite the long second query in Section 4 using SQL-MapReduce:

```
SELECT  PROD_DESC1, PROD_DESC2, PROD_DESC3, COUNT(*) AS CNT
FROM    BASKET_GENERATOR(
        ON (SELECT  SF.STORE_ID, SF.REG_ID, SF.TRAN_NO, SF.ITEM_ID,
                  SF.DT, PD.PROD_DESC, PD.PRICE
           FROM    SALES_FACT SF INNER JOIN PRODUCT_DIM PD
                  ON SF.ITEM_ID = PD.ITEM_ID) AS TRANSACTIONS A
        PARTITION BY STORE_ID, REG_ID, TRAN_NO, DT
        BASKET_ITEM('PROD_DESC')
        BASKET_SIZE('3'))
GROUP BY PROD_DESC1, PROD_DESC2, PROD_DESC3
HAVING  COUNT(*) > 1000
ORDER BY COUNT(*) DESC
```

Again, this query is a lot simpler, and it must be obvious that it will be easier to improve the performance of this query. The function `BASKET_GENERATOR` being used is a function specifically designed for market basket analysis. It makes it easier to formulate the query and even more importantly, all the processing required by `BASKET_GENERATOR` is done in parallel, offloading almost all the analytical processing to the Workers.

Built-in Functions – Aster Database offers an extensive set of built-in functions for statistical, relational, path, and pattern matching, graph, and text analysis (see Appendix A at the end of this paper for the list of functions currently supported).

11 Option 5: Use Apache's Hadoop

Apache's Hadoop is a software framework designed for supporting data-intensive applications. It's for those applications where a continuous stream of new data has to be stored and managed, and where all that data has to be analyzed periodically. It could be a click stream application generating enormous amounts of records, or a sensor-driven application (RFID-based) that needs to store a continuous stream of measurements. Some of these applications generate thousands of new data records per second. All this data needs to be stored for future use, leading to a massive amount of data storage. Another strength of Hadoop is that it supports queries that need to analyze large portions of all the data. Because of this feature, Hadoop is regularly positioned for advanced analytics, and therefore described in this section.

Architecture of Hadoop – Hadoop consists of a large number of modules. We briefly introduce the core modules here. For a more extensive description, we refer to Tom White's book⁸ on Hadoop.

- The foundation of Hadoop is formed by *HDFS* (Hadoop Distributed File System) which is responsible for storing and retrieving data. It's designed and optimized to deal with large amounts of incoming data per time-unit and for managing enormous amounts of data up to the petabytes.
- Like Aster Database, Hadoop is based on Google's MapReduce programming model, which allows for distributing data inserts and data access over hundreds of nodes.
- The module called *HBase* is used when random, real-time read/write access is needed. It operates on top of HDFS.
- The module called *Hive* offers a SQL-like interface for querying data. It supports a dialect of SQL called *HiveQL*. HiveQL supports the more classic features of SQL, but is missing some, such as subqueries in the `SELECT` and `FROM` clause are not allowed, and the `HAVING` and `LIMIT` clause are not supported.

⁸ White, Tom, 'Hadoop, 'The Definitive Guide'', Yahoo Press, 2010, second edition.

- Next to Hive, developers can also *Pig* for querying the data. The language supported by this module is called Pig Latin and is of a lower level than HiveQL. In fact, Pig Latin consists of a set of functions that slightly resemble the operators of SQL, such as group by, join, and select.

Characteristics of Hadoop – Any request for data, whether it's specified in HiveQL, Pig Latin, or any other interface, is executed in the form of a batch job. Each such job is scheduled and distributed over as many nodes as possible. In the background the process of starting and running all those jobs is monitored and managed by Hadoop. Distributing and managing all these jobs, requires additional processing. However, considering the amount of data analyzed, this overhead of additional processing may be acceptable if the user does not need to interact between query and result with minimal delay. Some database servers would not even be able to query so much data. This extra time for management is the penalty paid for being able to analyze these huge amounts of data and still get a decent performance.

Hadoop uses a so-called key/value structure for storing data. This implies that values making up a logical record are stored as separate physical records. This makes the data structure very flexible. New columns can easily be added on the fly. In fact, different records of a table can have different sets of columns. This storage format is also suitable for storing structured data *and* large unstructured data values, such as audios, videos, and documents.

Currently, most applications developed with Hadoop do not yet use HiveQL, but one of the other low-level APIs. These APIs are neither declarative nor storage independent, which will have a negative impact on productivity and maintenance; see Section 4. On the other hand, the advantage is that developers, who know the technical characteristics of Hadoop, can fully exploit the power of Hadoop.

Comparison of Aster Database and Hadoop – On a high abstraction level, Hadoop has a lot in common with Aster Database. Both use Google's MapReduce programming model, both can manipulate large quantities of data, both support SQL, and both are good at analyzing structured and semi-structured data. But, as indicated, notable differences do exist as a result of which they are applicable for different application areas.

Hadoop is designed for environments with the following characteristics:

- Where operational applications continuously generate massive amounts of structured, semi-structured, and unstructured data per time unit.
- Where all that generated data has to be stored fast and safely.
- Where the need exists to periodically analyze large portions of the data without having to copy it first to a separate database server (possibly a data warehouse).

Aster Database, on the other hand, is designed for environments with these characteristics:

- Where structured and semi-structured operational data is periodically copied from operational systems to the Aster Database using ETL-, or ELT-like solutions.
- Where large amounts of data are analyzed using a combination of SQL and complex forms of analytics.
- Where the need exists to support all forms of interactive analytics, allowing users to go back and forth between queries and results with a minimum of delay.

Table 1 contains a somewhat technical comparison of the two. As indicated, Hadoop developers can choose which API they prefer to use to query the data, therefore, this table contains a column for each of those APIs to show the differences.

	Aster Database	Hadoop: MapReduce + Hive	Hadoop: MapReduce + HBase	Hadoop: MapReduce + Pig	Hadoop: MapReduce native interface
Support for MapReduce	Yes	Yes	Yes	Yes	Yes
Accessible by most reporting and analytical tools	Yes	Yes	No	No	No
Familiarity of interface to BI developers	High	High	Low	Low	Low
Declarative and storage independent programming interface	Yes (SQL)	Yes (SQL with restrictions)	No	No (SQL-like)	No
Suitable for interactive form of analytics	Yes	No	No	No	No
Built-in reporting and statistical functions	Yes	No	No	No	No
Who or what acts as optimizer	Aster Database (cost-based optimizer)	Hive (rule-based optimizer)	Developer	Developer	Developer
Difficulty of developing user-defined functions	Easy	Complex	Complex	Complex	Complex
Comes with integrated development environment	Yes	No	No	No	No
Database administration concepts familiar to data warehouse administrators	Yes	No	No	No	No

Table 1 *Technical comparison of Teradata's Aster Database and the various interfaces supported by Apache's Hadoop*

Summary – The Aster Database and Hadoop are both more than qualified for processing and analyzing massive amounts of data. The strength of Hadoop is the combination of two characteristics: being able to process and store large amounts of incoming data, plus being able to query large portions of the data periodically. Hadoop is currently not ideal for interactive analytics where users can go back and forth between queries and results, and where they expect instantaneous results. The strength of Aster Database, on the other hand, is that it can manage large amounts of stored data, plus it supports all forms of analytics on those large amounts of data, including those forms where users interactively analyze data. In addition, Aster Database comes with a suite of 50+ pre-packaged analytic modules and an integrated development environment to help analysts and data scientists be productive more quickly.

12 Technical Advantages of SQL-MapReduce

This section lists the technical advantages of SQL-MapReduce for advanced analytics. The next section describes the business advantages of SQL-MapReduce.

Parallelization of Complex Operations – Operations that are hard to parallelize by most database servers, such as joins, group-bys, complex calculations, and operations that are non-relational by nature including most of the time-series based operations, can be implemented inside MapReduce functions. The processing of these functions is always parallelized.

Simplification of Queries – Power users and report developers don't have to concern themselves with the internal workings of MapReduce. This simplifies the writing of many analytical queries. They only have to study what the parameters of the MapReduce functions mean.

Efficiency of Low-level Programming Language – MapReduce functions are coded in a low-level programming language, such as Java, C++, C#, Python, and R. The low-level programming code is compiled (when it concerns languages such as Java and C++) and therefore executes very efficiently. No optimizer is needed to try and come up with the best processing strategy for the function code.

Efficient Data Access – Instead of using SQL statements or so-called cursors, the function code applies to one row and is activated for each row separately or applies to a partition. The main advantage of the row-by-row and partition-by-partition approaches is they are very efficient and improve query performance. This efficiency is independent of how data is stored on disk (i.e. it applies to row-stores, column-stores, object-stores, etc.). The programmers can determine how efficient the code is.

Predictable Query Performance – Because function processing normally requires a fixed amount of processing time for one row, a large proportion of query processing is predominantly function processing the number of rows and the number of nodes that determine the performance. This makes the query performance very predictable. For example, doubling the number of rows, probably increases the performance with a factor of two.

Linear Scalability – Due to predictable performance, the environment scales almost linearly. For example, doubling the number of nodes and partitions could improve the performance with a factor of (close to) two.

Extensive Set of Built-in Functions – As indicated, developers can create their own MapReduce functions. But Aster Database also comes with a large set of powerful, built-in functions for various forms of statistical, path, and relational analysis; see Appendix A.

Polymorphism of the Functions – If functions are coded correctly, they are *polymorphic*. This means the code can be written independent of the tables and columns being accessed. It's only when a function call is shipped to the Workers and right before it's executed, that the code is linked to the correct tables and columns. This is a form of late binding. The advantage is that the same type of function doesn't have to be written for every table and column. For example, a

function can be written that determines the top ten values of a column and it can be invoked for every column of every table. In fact, the function `GET_NEXT_FLIGHT_1HR` is polymorphic because other table and column names can be specified as parameters. And the built-in functions are all polymorphic too.

Polymorphism should not be confused with the concept of *overloading* where different functions with the same name (but with different parameters or parameter data types) can be developed. The advantage of polymorphism is improved productivity and maintenance.

Nesting of the Functions – All the MapReduce functions can be nested, meaning the result of one function can be passed to the next. Thus, the following query would be allowed. Here the result of `MR-FUNCTION1` is passed to `MR-FUNCTION2`:

```
SELECT *
FROM MR-FUNCTION2(ON SELECT *
FROM MR-FUNCTION1(...))
```

Note: The concept of nesting is widely used in SQL—queries, scalar functions, and views can all be nested. So the ability to nest MapReduce functions fits well with the language.

Comments on the SQL-MapReduce option – SQL-MapReduce offers some very valuable advantages for advanced analytics. But there are a few considerations. One is that some developers must learn how to write these MapReduce functions. But if a developer has experience with one of the more modern programming languages, such as Java or C, the learning process should be short. Note that developing new functions is done primarily by a small group of specialists, not by the whole community of SQL developers. Once developed, the SQL-MapReduce functions can then be easily invoked via standard SQL and used by analysts and BI tools without any procedural programming knowledge.

A second comment is that, although the syntax for invoking the MapReduce functions is according to the syntax of the so-called *window functions* defined in the SQL standard, the SQL code to invoke the functions is currently not portable.

13 Business Advantages of SQL-MapReduce

For the business itself, it doesn't matter which database language is used to support a specific analytical tool. Whether it's pure SQL, SQL-MapReduce, or MDX, business users won't see a language but instead, a highly graphical and user-friendly interface that hides the language. So while SQL-MapReduce doesn't offer any direct advantages to users, it does offer indirect advantages. Combining SQL with MapReduce creates a platform for advanced analytics and offers the following business advantages:

- If queries are executed more quickly, more detailed data can be analyzed. Improved query performance opens the doors for *deep and big data analytics*: storing and accessing massive amounts of detailed data with an acceptable performance.

- If queries are executed more quickly, more queries can be processed and more complex queries can be executed in the same timeframe. The former means users can execute more reports and more users can be supported. The latter implies support for *complex analytics*. Complex statistical models can be researched and data mining models can be created. So a larger portion of decision makers can exploit and benefit from the same investment made for the current business intelligence environment.
- If queries are executed more quickly, a more simple business intelligence architecture can be implemented. There will be less need for additional data stores in the architecture. And the fewer the data stores, the easier it is to change the architecture if user information needs change—less is more and more flexible.
- If queries are executed more quickly and if it leads to a simplification of the business intelligence architecture, the data can be 'moved' more quickly from the operational databases via the data warehouse to the reports. This means *operational analytics* on a large central data warehouse becomes an option.
- If queries are executed more quickly, there is less need to optimize and tune the data warehouse environment. Which means more ad hoc or unplanned queries can be processed with an acceptable performance. This is ideal for *self-service analytics* and even for self-service analytics on detailed data.
- If queries are executed more quickly, it's possible those queries that were forbidden to be executed because they consume too much system resources, might now be allowed. The negative consequence of forbidding queries is that the analytical capabilities are restricted. With SQL-MapReduce, those queries might be executed without disturbing the rest of the system.

14 Case Studies

This section contains two brief case studies of organizations using Aster Database with SQL-MapReduce.

Eightfold Logic – Eightfold Logic (formerly Enquisite) is an inbound marketing innovator whose mission is to empower businesses to attract and engage the highest possible volume of qualified website visitors, with the lowest possible marketing programs investment - optimizing the marketing return-on-investment (ROI). As marketers discover how inbound marketing channels are delivering significantly lower cost-per-lead compared to traditional outbound marketing channels, search engine, and social marketing optimization efforts are shifting the traditional program-heavy outbound marketing mix dramatically in favor of data-driven marketing optimization applications.

Eightfold Logic collects a large volume of web traffic data with Teradata's Aster Database as the central analytic platform to produce trend analysis of web traffic for clients. Previously, another database server was being used but this one had performance and scaling issues. The data in Aster Database is refreshed once per hour and it takes less than 5 minutes to transform and insert all the new data. This provides clients with a real-time, ad hoc analysis environment.

A query that used to take 45 minutes now runs in less than seven seconds. A complete scan of the database used to take 30 hours but now takes under 40 minutes. There are no problems with the mixed workload environment of queries and loading. They have developed nine new MapReduce functions. Currently their database grows at 12 to 15 gigabytes per day. The total database size is in the billions of rows and is over 4TB in volume.

MySpace – MySpace is an online community for people to discover and be discovered. Content creators can connect with fans, users can share and discover the latest videos and two people with similar interests can meet for the first time.

The data stored in MySpace's analytical data warehouse is used to understand the overall impact the site design has on their customers, as well as help forecast what changes will impact their customers. Their use of SQL-MapReduce within the web realm and social graph combined creates a special use case that other web companies wouldn't have to process. And with their volumes, MySpace is in the top percent of total data that anyone has to crunch through. Ultimately, using SQL-MapReduce is really the best solution for them.

Beside the use of Aster Data's *nPath*, they have developed four extra SQL-MapReduce functions. They have two clusters, one approximately 130TB large and another that is around 185TB in size. They keep a one month moving window of detailed data in each, which is about two trillion records in total for each cluster. Of the total these queries would run against, it would probably be on a table that has 87 billion records. Of those records, about three billion are a part of a SQL-MapReduce function.

Query times range from minutes to hours depending on the complexity of the process being performed. Because their main functions deal with iteration over a set, there can be times when they are processing data within the set that has billions of records in it, and then other times where the same query may run for millions of records. For instance, some queries using the *nPath* function access hundreds of millions of records and return data in about five minutes. Whereas, their user-defined *TraVersal* function may run over the same set of data, but take only a couple of hours. This might seem slow, but their feeling is that running these same queries on other platforms would likely be impossible.

15 Conclusion

The need for storing more detailed and more operational data in data warehouses is growing. In addition, the reporting and analytical workload is growing, and queries are becoming more complex. These new demands require new data warehouse platforms. Different platforms are available, but one with the most promising potential was created by merging SQL and MapReduce. It allows the processing of large analytical parts of a query to be pushed close to where data is stored without losing SQL's declarative and storage independency properties. Although the combination of SQL and MapReduce is purely a technical issue, the impact on the business and decision making is considerable.

About the Author Rick F. van der Lans

Rick F. van der Lans is an independent analyst, consultant, author, and lecturer specializing in data warehousing, business intelligence, service oriented architectures, and database technology. He works for [R20/Consultancy](#), a consultancy company he founded in 1987.

Rick is chairman of the annual European Data Warehouse and Business Intelligence Conference (organized in London), chairman of the [BI event](#) in The Netherlands, and he writes for the [B-eye-Network](#). He introduced the [Data Delivery Platform](#), a new architecture for developing more flexible business intelligence architectures.

He has written several books on SQL. His popular [Introduction to SQL](#) was the first English book on the market in 1987 devoted entirely to SQL. After more than twenty years, this book is still being sold, and has been translated into several languages, including Chinese, German, and Italian.

- Introduction to SQL, fourth edition
- SQL for MySQL Developers
- The SQL Guide to SQLite
- The SQL Guide to Ingres
- The SQL Guide to Pervasive PSQL
- The SQL Guide to Oracle

He has been a member of the standardization committee responsible for the SQL standard for seven years.

For more information, please visit www.r20.nl, or email to rick@r20.nl.

About Aster Data

Teradata's Aster MapReduce Platform is the market-leading big data analytics solution. The Aster analytic platform embeds MapReduce analytic processing for deeper insights on new data sources and multi-structured data types to deliver analytic capabilities with breakthrough performance and scalability. Aster's solution utilizes Aster's patented SQL-MapReduce to parallelize the processing of data and applications and deliver rich analytic insights at scale. For more information visit www.asterdata.com or for more about Teradata, see teradata.com.

About Teradata

Teradata is the world's leader in data warehousing and integrated marketing management through its database software, data warehouse appliances, and enterprise analytics. For more information, visit teradata.com.

Appendix A The Built-in Functions of Aster Database

This appendix contains a list of functions supported by the MapReduce System. It shows the richness and the extensiveness.

Area	Analytics
Path and Pattern Analysis Discover patterns in rows of sequential data	nPath: complex sequential analysis for time series analysis and behavioral pattern analysis nPath Extensions: count entrants, track exit paths, count children, and generate subsequences Sessionization: identifies sessions from time series data in a single pass over the data
Statistical Analysis High-performance processing of common statistical calculations	Histogram: function to assign values to bins Decision Trees: function for creating a model of decisions and their possible implications Approximate percentiles and distinct counts: calculate percentiles and counts within specific variance Correlation: calculation that characterizes the strength of the relation between different columns Regression: performs linear or logistic regression between an output variable and a set of input variables Averages: calculate moving, weighted, exponential or volume-weighted averages over a window of data GLM: generalized linear model function that supports logistic, linear, log-linear regression models. Returns all parameters similar to R/SAS * Naïve Bayes Classifier: simple probabilistic classifier that applies Bayes Theorem to data sets. * Support Vector Machines: a supervised learning method used for classification and regression analysis * PCA: Principal Component Analysis transforms a set of observations into a set of uncorrelated variables *
Graph and Relational Analysis Analyze patterns across rows of data	Graph analysis: creates configurable groupings of related items from transaction records in single pass nTree: finds shortest path from a distinct node to all other nodes in a graph Other: triangle finding, square finding, clustering coefficient *

Table 2 Examples of built-in functions supported by Aster Database (Source Teradata); the functions identified with * are new as of September 2011 – Continuous on the next page

Area	Analytics
Text Analysis Derive patterns in textual data	Text Processing: counts occurrences of words, identifies roots, and tracks relative positions of words & multi-word phrases nGram: split an input stream of text into individual words and phrases Levenshtein Distance: computes the distance between two words Sentiment Analysis: classify content is positive or negative (for product review, customer feedback)* Text Categorization: used to label content as spam/not spam * Entity Extraction/Rules Engine: identify addresses, phone number, names from textual data *
Cluster Analysis Discover natural groupings of data points	k-Means: clusters data into a specified number of groupings Canopy: partitions data into overlapping subsets within which k-means is performed Minhash: buckets highly-dimensional items for cluster analysis Basket analysis: creates configurable groupings of related items from transaction records in single pass Collaborative Filter: predicts the interests of a user by collecting interest information from many users
Data Transformation and utilities Transform data for more advanced analysis	Unpack: extracts nested data for further analysis Pack: compress multi-column data into a single column Antiselect: returns all columns except for specified column Multicase: case statement that supports row match for multiple cases Pivot: convert columns to rows or rows to columns * Log parser: Generalized tool for parsing Apache logs *

Table 3 Examples of built-in functions supported by Aster Database (Source Teradata); the functions identified with * are new as of September 2011 – Continuation of the previous page