Discovering Business Insights in Big Data Using SQL-MapReduce®

A Technical Whitepaper

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

July 2013

Sponsored by
TERADATA®

Copyright © 2013 R20/Consultancy. All rights reserved. Teradata, the Teradata logo, Aster, SQL-MapReduce, Applications-Within, SQL-H are trademarks of registered trademarks of Teradata Corporation and/or its affiliates in the U.S. and worldwide. Trademarks of companies referenced in this document are the sole property of their respective owners.

Table of Contents

1 Summary 1

2 Discovery and the Data Scientist 3

3 The Discovery Process 5

4 Requirements for a Discovery Platform 7

5 Technologies for Implementing a Discovery Platform 9
5.1 The World of Big Data, NoSQL, and Schemas 10
5.2 Using SQL for Discovery 13
5.3 Functions in SQL 17
5.4 Parallelization of SQL Queries and SQL Functions 19
5.5 Hadoop and MapReduce in a Nutshell 24
5.6 The Marriage of SQL and MapReduce: SQL-MapReduce 27

6 Implementing a Discovery Platform 30
6.1 Solution 1: Classic SQL System 31
6.2 Solution 2: Advanced Reporting and Analytical Platform 31
6.3 Solution 3: SQL-MapReduce System 32
6.4 Solution 4: Hadoop with MapReduce 33
6.5 Solution 5: Hadoop with a SQL Interface 34

7 Comparison of Five Solutions for Implementing a Discovery Platform 35

8 Technical Advantages of SQL-MapReduce 37

About the Author Rick F. van der Lans 39

About Teradata and the Teradata Aster Discovery Platform 39

Appendix A – The Built-in Functions of Teradata Aster Database 41

Copyright © 2013 R20/Consultancy, all rights reserved.

1 Summary

This whitepaper describes the advantages of merging the openness and productivity of SQL with the scalability of MapReduce to create a ***discovery platform*** *that supports today's complex and data-intensive analytical workload generated by data scientists. It focuses on the SQL-MapReduce® implementation offered by Teradata through the Teradata Aster Discovery Platform which includes the Aster Database and Aster Discovery Portfolio. The whitepaper also discusses some of the alternative technologies available for developing a discovery platform, such as Hadoop, MapReduce, NoSQL, schema-on-read, and SQL-fication.*

Business intelligence users are traditionally classified based on the tools they use: users of reporting tools and users of analytical tools. But there is a third group of users, one that uses anything they can find to discover new insights that can lead to business benefits. They can benefit from reporting and analytical tools, but they need more, they need *discovery* capabilities. Discovery is about searching and analyzing data to find new business insights that can lead to business opportunities.

Discovery is about searching and analyzing data to find new business insights.

Nowadays, analysts responsible for discovery are called *data scientists*. Data scientists commonly use all the data and all the tools they can lay their hands on. They use analytics and reporting to study data, but they won't stop there. In other words, discovery is not a fancy new term for analytics. Analytics is just one of the many technologies used by a data scientist to discover new insights.

The *discovery process* commonly followed by data scientists consists of four steps: data acquisition, data preparation, data analysis, and data interpretation. To support data scientists in this discovery process, a reporting tool or analytical tool is not sufficient, they need a feature-rich, fast, and flexible *discovery platform*. Such a platform should minimally support the following features:

- Data scalability
- Heterogeneous data access
- Complex value analysis
- Data preparation techniques
- Multiple analysis techniques
- Multiple analysis tools
- Interactive analysis
- High-speed analysis
- High-level development language

Data scientists can select from different solutions for implementing a discovery platform:

1. Classic SQL system
2. Advanced Reporting Platform
3. SQL-MapReduce System
4. Hadoop with MapReduce
5. Hadoop with SQL interface

Copyright © 2013 R20/Consultancy, all rights reserved.

This whitepaper describes all five solutions in detail and focuses on the third one, the SQL-MapReduce® solution. SQL-MapReduce is a framework based on a combination of SQL, which is the most popular database language, and 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 the *Teradata Aster Database* (formerly called Aster Data *n*Cluster). Aster Database is part of the *Teradata Aster Discovery Platform* which also includes the *Teradata Aster Discovery Portfolio*.

On the outside, the Teradata Aster Database looks like any other SQL database server. It supports standard SQL and all the common APIs, such as ODBC and JDBC, so that it can be accessed by all the popular analytical and reporting tools. What's inside makes Aster Database special. The product has been designed specifically for discovery and exploration of big data with the intention to uncover business insights. Its unique Applications-Within™ architecture runs analytic application logic inside the database, leveraging its massively-parallel architecture and SQL-MapReduce to fully parallelize the processing of complex analytical queries.

Besides being a powerful platform for data scientists, the support for standard SQL makes Aster Database also suitable for more traditional query workloads such as reporting and analytics, thus making it a platform for *business analysts* as well.

To summarize, extending a SQL database server with MapReduce creates a discovery platform that combines the expressive query power, openness, and productivity of SQL with the parallelizability and scalability of MapReduce. The combination has the potential to improve the performance of complex analytical queries running on large to extremely large datasets. Teradata Aster Database is a mature and robust implementation of SQL-MapReduce and has proven itself as a discovery platform.

Teradata's Aster Database is a mature and robust implementation of SQL-MapReduce and has proven itself as a discovery platform.

Note: This whitepaper is a rewrite of an older whitepaper entitled *Using SQL-MapReduce for Advanced Analytical Queries*[1] and was published in September 2011. Since then, much has changed: the Hadoop stack has grown with several new modules, new SQL interfaces for HDFS have been released, new versions of Aster Database have become available, and the interest for big data has grown drastically. Therefore, it was decided to drastically rewrite this whitepaper. Some pieces of text have been reused, but major sections are new or have been completely revised.

[1] R.F. van der Lans, *Using SQL-MapReduce for Advanced Analytical Queries*, September 2011.

Copyright © 2013 R20/Consultancy, all rights reserved.

2 Discovery and the Data Scientist

Business Intelligence and Discovery – Boris Evelson of Forrester Research[2] defines business intelligence as follows:

> *Business intelligence is a set of methodologies, processes, architectures, and technologies that transform raw data into meaningful and useful information used to enable more effective strategic, tactical, and operational insights and decision-making.*

From this definition can be derived that business intelligence is not a tool, not a technology, nor some design technique, but it's everything needed to transform and present the right data in a form that leads to *insights* and improves the decision-making processes of an enterprise.

The tools used by decision makers to study and analyze data, can be classified in two main categories: *reporting tools* and *analytical tools*. In principle, reporting tools show what has happened. Although the shown data may have been transformed, processed, and aggregated, still, the data shows the past and current situation. Typical examples of questions answered with reporting tools are "Show the total revenue per sales region for the last two weeks" and "Present a 360° report of a particular customer." Dash boards are also examples of reports; OLAP tools with which users can look at data from every angle and at every level of detail, belong to this category as well, as do batch reports.

Analytical tools, on the other hand, are used to find out what may or can happen. They use techniques such as predictive modeling, simulation, and forecasting. The result of analytics is usually not (aggregated) data but a set of rules. Examples of such rules are "When a customer buys cola and chips, there is a 75% chance he buys dipping sauce as well" and "The most efficient route to deliver goods to a particular set of shops is the following."

> *Reporting tools show what has happened and analytical tools show what may or can happen.*

BI users can be classified based on the tools they use: reporting users and analytics users, where the former is usually the bigger group. But there is a third group of users, one that uses anything they can find to discover new insights which can lead to business benefits. They can benefit from reporting and analytical tools, but they need more, they need *discovery* capabilities.

Discovery is about searching and analyzing data to get some new business insights that can lead to business opportunities. Their questions are not that straightforward so that they can be answered by starting up a particular report or by firing up a pre-defined analysis. Examples of their questions are:

- What is a possible behavioral pattern of credit card usage that signifies a fraudulent action?
- What are other forms of data that can help us locate deeply buried oil fields more easily?

[2] B. Evelson, *Topic Overview: Business Intelligence*, November 21, 2008.

Copyright © 2013 R20/Consultancy, all rights reserved.

- How high is the financial risk if a person 21 years old with no job is given a mortgage?

The challenge for discoverers is that they don't always know exactly what they are searching for, although they probably have a feeling or an inkling.

The Data Scientist – Nowadays, we call these discoverers who try to gain knowledge or awareness of something not known before, *data scientists*. The data scientist has been called the sexiest job of the 21st century[3] by the Harvard Business Review. But what is a data scientist and what does he do? For example, in an oil company, the ones responsible for analyzing soil test results to locate new oil fields or for analyzing new techniques to find new oil fields faster, can be classified as data scientists. Another clear example of a data scientist is an actuary working for an insurance company. Actuaries deploy mathematics, statistics, and financial theory to analyze the financial consequences of risk. Professors looking for cures for specific diseases by doing DNA research can also be classified as data scientists.

Data scientists are discoverers who try to gain knowledge or awareness of something not known before.

Usually, data scientists use all the data and all the tools they can get their hands on. They use analytics and reporting, but they won't stop there. In other words, discovery is not a fancy new term for analytics. Analytics is just one of the many tools used by a data scientist to get new insights.

Although the term data scientist may be new, this profession has existed for a long time. For example, Napoleon Bonaparte used mathematical models to help make decisions on battlefields. These models were developed by mathematicians, Napoleon's own data scientists. Another (famous) example of that same time period is the Minard Map[4]. This is a good example of a data scientist using geo visualization to analyze data. The map depicts the advance into and retreat from Russia by Napoleon's army in 1812-1813. This army was practically destroyed during the retreat; the army left with 422,000 troops and came back with a mere 10,000. Charles Joseph Minard was clearly a data scientist. Many more examples like this can be found.

Data scientists are smart people. They need business knowledge, they need to understand the enterprise data, they need to know how to deploy technology, they have to understand statistical techniques, visualization techniques, and, most importantly, they need to know how to *interpret* the results. For example, if an analysis exercise shows that the number of storks born has a strong correlation with the number of babies born one year later, data scientists should have sufficient knowledge to conclude that these variables do not have a direct relation, but that they are both dependent on a third variable, one that probably hasn't been included in the study yet.

Data scientists need to understand enterprise data, technology, statistical techniques, visualization techniques, and they need to be able to interpret results.

The Data Scientist versus the Business Analyst – A traditional user of business intelligence systems is the *business analyst*. A business analyst assists end users in making informed business decisions. He exploits a data warehouse to uncover important facts and statistics that show an

[3] T.H. Davenport and D.J. Patil, *Data Scientist: The Sexiest Job of the 21st Century*, Harvard Business Review, October 2012.
[4] Wikipedia, see http://en.wikipedia.org/wiki/Charles_Joseph_Minard, May 2013.

Copyright © 2013 R20/Consultancy, all rights reserved.

organization's performance. He helps transform business needs in reports, he analyzes data structures, and defines business concepts. Quite often, he operates on the frontier between the IT department and the business departments.

Data scientists and business analysts may be using the same data, but they use that data differently. As indicated, the discovery work of a data scientist is about searching and analyzing data to produce new business insights that can lead to business opportunities. The work of the business analyst is more concrete. He creates reports for himself and for end users, he helps end users to develop their own reports, and so on.

The boundary between these two jobs is not as clear cut as one may expect. Business analysts may be doing data scientists work occasionally, and vice versa. In fact, the person working as data scientist today, may have the role of business analyst tomorrow.

3 The Discovery Process

The way data scientists work, is called the *discovery process*. In this chapter we list and describe the characteristics of this process.

A Four Step Process – The discovery process consists of four steps (see also Figure 1):

- **Data acquisition:** In this first step, data is collected from various data sources. The data scientist selects the data sources that may be useful for the study.

- **Data preparation:** In this step, data is transformed, aggregated, integrated, and cleansed until it has the form that data scientists want for their study. For example, for many data mining algorithms it can be useful to transform real life values to binary values.

- **Data analysis:** In this step, data is analyzed with various types of techniques, including simple reporting techniques; classic statistical techniques, such as forecasting, predictive modeling, and clustering; data mining techniques; data visualization techniques such as affinity visualization, path visualization, scatter clouds, geo-visualization techniques; and time-series analysis.

- **Data interpretation:** When the techniques and tools present results and insights, it's still the responsibility of the data scientist to determine whether the results make sense. This requires in-depth knowledge of the business and the data, and it demands common sense.

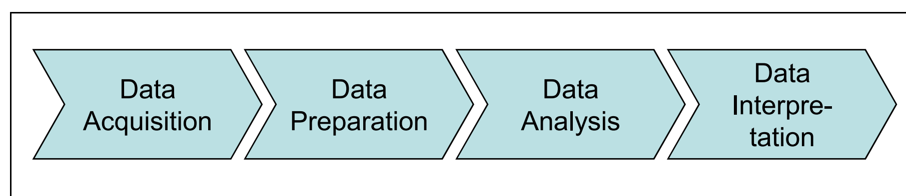


Figure 1 *The discovery process consists of four steps.*

Copyright © 2013 R20/Consultancy, all rights reserved.

The Result of Discovery – The result of a discovery process is in most situations *insights*, and these insights are formulated as a set of *rules*. These rules can be simple if-then rules, for example, if two payments are done with the same credit card within 10 seconds, they are probably fraudulent. Rules can also be advanced statistical formulas indicating the relationship between specific variables. For example, a 10 degree rise in temperature increases sales of barbecue meat with 300%. Sometimes rules are sophisticated, self-learning data mining models that can predict customer behavior by combining historical and new incoming data.

Spinoff Results – It's not uncommon that during the discovery process unexpected insights and rules are found. These spinoffs can be as useful as the rules intended to be found. Remember Alexander Fleming who discovered penicillin by accident. There are more well-known examples like this. For example, chemist William Perkin wanted to invent a cure for Malaria. His experiments lead accidently to the first-ever synthetic dye. And don't forget George Crum who discovered Coke by accident when searching for a cure for headaches.

No Clear Goal – Another characteristic that shows that data scientists are different from most other BI users, is that their analysis work doesn't always have a clear goal. The work they do is much more free format, much more research-like.
Because the goal is not always that clear, classifying this process as "finding a needle in a haystack", doesn't always make sense. If you're looking for a needle in a haystack, the goal is very clear, and with a good magnet it's not even that complex. Discovery is much more a stepwise refinement process. With each step, the data scientist may get closer to useful insights.

Wide Range of Analysis Techniques – As indicated, data scientists use a wide range of analysis techniques to discover new insights. Many well-known statistical techniques can be used to find rules. A data scientist should have access to all the tools and techniques he needs. He should also be able to mix and match them. For example, he may want to apply a time-series analysis first, followed by a geo-visualization of the result. Data scientists should not be restricted in discovering valuable insights due to the lack of tools and techniques.

Data Overload Doesn't Exist – The more data a data scientist has access to, the more discovery options he has. In this context, more means three things. First, it means more detailed data—no aggregate data. Aggregation of data can hide potential insights. Dealing with detailed data is a typical aspect of the big data trend. Nowadays, the technology exists to process massive amounts of data fast.
Second, more means more data sources. Having access to a data warehouse is probably not enough for data scientists. They also need access to large files with sensor data, spreadsheet data, external data sources, and so on. It wouldn't be the first time that rules are discovered by enriching internal business data with external data.
Third, more means more types of data. Giving data scientists access to structured data is very useful, but not all the data has a very rigid structure. Data scientists may also require access to what some call unstructured, multi-structured, semi-structured, or poly-structured data.

Data Scientists are Creators of Data – Usually, users of reporting tools don't create their own data. They access data stored in a data warehouse or data mart. In some situations, it could be that the data the data scientists need, doesn't even exist yet. The consequence can be that dedicated projects must be initiated to create and collect the required data. An interesting example of such a project is the *Amsterdam Born Children and their Development* (ABCD)

Copyright © 2013 R20/Consultancy, all rights reserved.

project. This project started in 2001 and still continues. The project tracks the health of 8,000 children. Every so many years these children have a checkup. The goal of this long lasting study is to discover what the relationship is between early growth and development on the overall health later on in live. This study is a good example of where the right data has to be created first.

The Discovery Process is an Iterative Process – Figure 1 suggests that the discovery process is a serial process: when one step is finished, the next one starts, and we never return to a previous step. However, less would be closer to the truth, the discovery process is very iterative. For example, when a data analysis step has been finished, the conclusion may be to collect more data, and start all over again. Even a data preparation step may lead to a return to the data acquisition step. In fact, this entire four-step process may have to be repeated several times before the right insights rise to the surface.

Long Lasting Discovery Projects – Some discovery processes are completed in one day, but they can also last for weeks, months, and even years. For example, in April 2013, researchers working at the academic hospital in the city of Utrecht in The Netherlands discovered a formula that predicts for patients, who have had an heart attack or stroke, the risk of new health problems ten years later. The formula looks at fourteen variables, including age, gender, smoking habits, and blood pressure. This study started in January 1996 and ended in 2013. This is a good example of long lasting discovery projects.

Actionable Discovery Results – When a discovery process is finished, the organization has experienced no advantages yet—no money has been made, no ROI. The discovery process has to be followed up by a step called *Act*. In this step, the gained insights have to be used or implemented. Examples of implementing insights are: organization policies are changed, decision rules are embedded in operational applications, business processes are optimized, customers are offered special discounts, and so on. Without the Act step, the entire discovery exercise has been for nothing. In other words, it's important that discovery results are *actionable*. Note that the data scientist is not always involved in the Act.

4 Requirements for a Discovery Platform

To support data scientists in their discovery process, a reporting or an analytical tool is not sufficient. They need a feature-rich, fast, and flexible *discovery platform* that assists them with all four discovery steps. Such a platform should at least support the following features:

A discovery platform should offer data scalability.

Data Scalability – Many traditional information systems store and manage large numbers of records. The last years, new applications have been developed that store amounts of data magnitudes larger than those in the more traditional applications. For example, click-stream applications, sensor-based applications, and image processing applications all generate massive numbers of records per day. The amount of records stored surpasses more often than not hundreds of millions of records.
Nowadays, the popular term used for such systems is *big data systems*. Because these systems store data on such a detailed level, new insights that have always been hidden, become visible. Therefore, a discovery platform should allow data scientists to analyze big data fast and

Copyright © 2013 R20/Consultancy, all rights reserved.

efficiently. In other words, a discovery platform should offer *data scalability*. Data scalability[5] is the ability of a system to store, manipulate, analyze, and process ever increasing amounts of data without reducing overall system availability, performance, or throughput.

Heterogeneous Data Access – Most users of BI systems find the data they need in the enterprise data warehouse or in one of the data marts. Not so for data scientists. The data they need can be hidden in numerous data stores of which the data warehouse is probably one. But they may also want to include data from external websites in their analysis research, results from their own tests and studies, textual data in documents, and so on. Data scientists are *data-greedy*. For them the rule applies: more is better, because by being able to analyze more data, more valuable insights can reveal themselves. Therefore, to support the data acquisition step, a discovery platform should make it easy to access multiple data stores, including data stores using different technologies. In addition, the platform should allow the mixing and matching of data from this heterogeneous set of data stores.

Data scientists are data-greedy.

Analysis of Complex Values – Most data values stored in SQL database servers are simple numbers, strings, and dates. But in more and more systems, data values are *complex values*, such as weblog entries, EDIFACT message, text documents, and audio streams; see also Section 5.1. Obviously, these data values do have structure, but that structure is part of the value itself. The result is that the database server isn't aware of that structure.
The amount of complex values that organizations store is increasing. If not already, analytics of large sets of complex values will be on everyone's agenda in the near future. It's important that a discovery platform allows data scientists to analyze such complex values.

A discovery platform should be able to analyze complex values.

Data Preparation Techniques – When the required data sources have been identified and have been made available for the data scientists, the data must be turned into a form that makes analysis more successful. In other words, *data preparation* must take place before analysis can start. Therefore, it's important that a discovery platform supports many features to integrate all the data, transform it, cleans it, filter it, and so on. For complex values, data preparation means that a structure must be assigned to these values before they are analyzed.

Multiple Analysis Techniques – As indicated in Chapter 3, data scientists should be able to use a multitude of analysis techniques, including simple reporting techniques; classic statistical techniques, such as forecasting, predictive modeling, and clustering; data mining techniques; data visualization techniques such as affinity visualization, scatter clouds, geo-visualization techniques; and time-series analysis. Therefore, to support the data analysis step, a discovery platform should offer an wide-range and integrated set of analysis techniques.

A discovery platform should offer an wide-range and integrated set of analysis techniques.

Multiple Analysis Tools – Usually, data scientists use all kinds of tools to analyze data, ranging from more classical statistical tools to visualization tools. A discovery platform should allow data

[5] Eugene Ciurana, *Getting Started with NoSQL and Data Scalability*, see http://refcardz.dzone.com/refcardz/getting-started-nosql-and-data

Copyright © 2013 R20/Consultancy, all rights reserved.

scientists to deploy all these different tools on the same data. They should be able to mix and match features of the analysis tools. For example, they may want to deploy a geographical grouping of data before it's passed to a statistical analysis. Or, they want to sessionize weblog entries, then deploy a sentiment analysis, and finally, use a visualization technique. In addition, it should not be needed to create replicas of the data for each and every tool. This would raise project costs and would slow down the discovery process too much. In an ideal situation, many of the results created with one tool should be reusable by another tool.

A discovery platform should support interactive analysis.

Interactive Analysis – The discovery process is not a serial process, but a highly iterative one; see Chapter 3. Therefore, a discovery platform must make it possible to run an analysis, change specifications, run the analysis again, and so on. A discovery platform can do this by supporting *interactive analysis*: when the data scientist wants to run a new query, he should be able to execute it right away. No scheduling of queries should be needed. The platform should not make data scientists wait too long between queries.

Out-of-the-box a discovery platform should offer high-speed analysis.

High-Speed Analysis – It's evident that a discovery platform offers high-speed analysis. Even if big data sets are analyzed using complex analysis algorithms, results should be displayed within seconds (and preferable faster than that). The faster an analysis is completed, the more an alternative hypothesis can be tested.
If analysis of a potential correlation between two variables requires two hours of processing, the data scientist will probably only analyze the variables he thinks are of interest. If such an analysis only takes a few seconds, he may want to analyze them all. This increases the chance that valuable results are discovered.
In addition, when performance is slow, a data scientist can lose his train of thought.
It's also important that a discovery platform offers high-speed analysis out-of-the-box. In other words, it should not be needed to first tune and optimize the database and database server before an analysis can start. Again, this would delay the discovery process too much.

A discovery platform should offer a high-level development language.

High-Level Development Language – A high-level development language is important for productivity and for the iterative nature of discovery. Data scientists should not be slowed down in their thinking process by the use of a low-level development language. In addition, it's important that existing developments can be reused easily.

5 Technologies for Implementing a Discovery Platform

Nowadays, data scientists can choose between many different technologies and products to implement a powerful discovery platform. For example, they can use a classic SQL system, an advanced reporting and analytical environment, a SQL-MapReduce based system, a NoSQL system, or a mix of these solutions. Before we can answer the question what the best discovery platform is, some of these technologies are described in this chapter. Chapter 6 addresses the question what the best discovery platform is.

Copyright © 2013 R20/Consultancy, all rights reserved.

5.1 The World of Big Data, NoSQL, and Schemas

One of the hottest trends in the IT industry is undoubtedly *big data*. With big data comes a new generation of NoSQL systems for data storage, such as Hadoop, Cassandra, and MongoDB. One of the key differences between NoSQL systems and the more classic SQL systems is the way they treat *schemas*. A schema describes the structure of data. Because schemas are an important technological concept for evaluating discovery platforms, this section explains the different styles of handling schemas.

Big Data and Data Scientists – Big data is a blessing for data scientists, because it gives them data on a level of detail that may reveal insights that would be impossible to see with aggregated or condensed data. Imagine a utility company that measures electricity consumption once a month. By deploying big data systems, that same company can switch to measuring consumption every few seconds. Because of this higher level of detail, insights may become visible that have always been completely hidden.
The interest of big data has always existed, even before the term was invented. However, most of the big data needs were impossible to implement with older technology or unaffordable. What has changed and what has given big data its position in the spotlights, is that hardware and software technologies have become available that have been designed specifically to support big data environments for reasonable prices.

NoSQL – Many of the new systems for big data storage are referred to with the intriguing term *NoSQL*. The reason this name has been selected is simple: most of them do not support SQL nor the relational model for managing data. For example, the group NoSQL systems called *document stores* uses a more hierarchical model for organizing data, and the *column-family stores* support tables in which each record can have a different set of values. What distinguishes them most from the well-known SQL systems is how they handle *schemas*.

The Schema – In SQL systems, when a table is created, a schema is assigned. A schema describes the structure of all the records in the table. It indicates how many columns there are, what the names of those columns are, what the data types are, which columns have unique values, and so on. All records of a table in a SQL system have the same schema. In a way, all records of a table inherit the schema of that table.

There are two different ways for data storage systems to handle schemas, these are called schema-on-write and schema-on-read. Both are described in this section.

Schema-on-Write – Classic SQL systems support *schema-on-write*. This means that all the data written to a database has a schema. For each value of each record is known to which column of the table it belongs. A schema is not optional.

With schema-on-write all data written to a database has a schema.

Not only SQL systems use schema-on-write. Older, so-called pre-relational databases, such as IMS and IDMS, also support schema-on-write. In addition, when data is stored as XML documents, schema-on-write is used as well, because the schema of the data is known at the time of writing. The advantage of schema-on-write is that when an application accesses the data, the schema is known, and therefore it doesn't have to apply application logic to assign a schema to the data during access.

Copyright © 2013 R20/Consultancy, all rights reserved.

Two forms of schema-on-write can be identified: fixed and variable.

In SQL systems all records in a table have the same schema. In fact, it's impossible for a SQL table to contain one row with ten columns and another row with twelve columns. This also means that if a new column has to be added to one record, that column must be added for every record. We refer to this form as *fixed schema-on-write*.

The alternative to fixed schema-on-write is *variable schema-on-write*. Several NoSQL systems use this. When data is stored in their databases, a schema in XML, JSON, or BSON form is written together with the data itself, thus schema-on-write is used. However, different records in one and the same table (or an equivalent concept) can have different schemas. In other words, the schema of each record in a table varies.

The advantage of variable-schema-on-write is flexibility. When a new column or element has to be added to only one record, it only has to be done for that one record. The other remain unchanged. No resource intensive operation has to be invoked to reorganize all the other records in the table as well.

Schema-on-Read – The opposite of schema-on-write is *schema-on-read*. When a database server uses schema-on-read the data has no schema when it's stored. Or, more precisely, the database server doesn't know what the schema is, the data values are like blobs of bytes. For example, the following long string of comma-separated values is a value without a schema:

> *With schema-on-read, stored data has no schema, the schema is assigned when reading the data.*

```
"Anchorage Daily News","PO Box 149001","Anchorage","AK","99514-9001","907-257-4200",
"907-258-2157","71","","82","http://www.adn.com/",newsroom@adn.com
```

If this string is stored as one value in a column of a SQL table, the database server doesn't understand the structure of this value and will treat it as one atomic value. Such values are called *complex values* or *schema-less values*. Both terms are used interchangeably in this whitepaper.

An EDIFACT message representing an invoice (the `XXX` code is used to separate elements) is a more complex example of a schema-less value:

```
UNB+UNOA:1+005435656:1+006415160:1+060515:1434+00000000000778'XXXUNH+00000000000117+INVOIC:D:97B:UN
'XXXBGM+380+342459+9'XXXDTM+3:20060515:102'XXXRFF+ON:521052'XXXNAD+BY+792820524::16++CUMMINSMIDRANG
EENGINEPLANT'XXXNAD+SE+005435656::16++GENERALWIDGETCOMPANY'XXXCUX+1:USD'XXXLIN+1++157870:IN'XXXIMD+
F++:::WIDGET'XXXQTY+47:1020:EA'XXXALI+US'XXXMOA+203:1202.58'XXXPRI+INV:1.179'XXXLIN+2++157871:IN'XX
XIMD+F++:::DIFFERENTWIDGET'XXXQTY+47:20:EA'XXXALI+JP'XXXMOA+203:410'XXXPRI+INV:20.5'XXXUNS+S'XXXMOA
+39:2137.58'XXXALC+C+ABG'XXXMOA+8:525'XXXUNT+23+00000000000117'XXXUNZ+1+00000000000778'
```

Someone familiar with the structure of EDIFACT messages knows what all these codes mean, but for the database server this is just one large value.

Copyright © 2013 R20/Consultancy, all rights reserved.

The following example of a schema-less value is a record coming from a weblog:

```
datestamp ip request 6/1/2012 11:10:19 AM 107.1.187.170 GET /x.php?u=http://studio-
5.financialcontent.com/synacor?Page=QUOTE&Ticker=DDD HTTP/1.1 6/1/2012 5:53:49 AM 107.1.2.180 GET
/tv/3/player/vendor/Chef%20Tips/player/fiveminute/content/steak/asset/gnrc_15879500 HTTP/1.1
6/1/2012 8:55:54 AM 107.34.51.63 GET
/tv/3/search/content/The%20Andy%20Griffith%20Show/s/The%20Andy%20Griffith%20Show HTTP/1.1 6/1/2012
3:12:43 PM 107.5.115.117 GET
/tv/3/search/content/Kathie%20Lee%20Gifford's%20epic%20'Today'%20gaffe/s/Kathie%20Lee%20Gifford's%2
0epic%20'Today'%20gaffe HTTP/1.1 6/1/2012 4:48:35 PM 108.225.132.245 GET
/tv/3/search/content/Deadliest%20Catch/s/Deadliest%20Catch HTTP/1.1 6/1/2012 10:25:12 AM
108.246.20.125 GET /x.php?u=http://studio-5.financialcontent.com/synacor?Page=QUOTE&Ticker=DJ:DJI
HTTP/1.1 6/1/2012 1:58:14 AM 108.246.25.117 GET
/tv/3/player/vendor/Chef%20Tips/player/fiveminute/content/steak/asset/gnrc_15879500 HTTP/1.1
```

Evidently, all three example values have a structure, but if these values are stored like above, the database server won't understand their structure, because it doesn't know its schema. There are more examples of large schema-less values that have a structure for which the database server doesn't know the schema, such as text blocks, audio, and video.

For applications to be able to process schema-less values, the data must be assigned a schema first. In other words, when the data is read, it should be assigned a schema, hence the name schema-on-read.

Schema-on-read is not limited to NoSQL systems. SQL systems support schema-on-read as well. Imagine a simple SQL table consisting of two columns: one containing some kind of identifier and the second one containing schema-less values, that's schema-on-read as well. Note that a table in a SQL system can have columns that hold values with a schema (schema-on-write) and other columns with schema-less values (schema-on-read).

Using schema-on-read has three advantages. One, it's flexible, because new data elements can be added to records without having to change the schema of the table. Second, loading of data is fast, because the incoming data doesn't have to be processed during the load process—no schema has to be assigned. The data is stored in its original form. Third, because the data hasn't been given a schema, the applications can change how they want to look at the data without having to change the table schema. The schema is determined when the data is read.

The main disadvantage of schema-on-read is that when data is retrieved, execution time has to be spend on assigning a schema to the data. Schema-on-read has two sub-forms:

- With *schema-on-application-read* it's the application that assigns a schema to the data. The schema-less data is retrieved by the database server from the data store and transmitted unchanged to the application. The application contains the logic that understands the structure and assigns a schema.

- With *schema-on-database-read* it's the database server that retrieves the schema-less data from the data store and, before it's transmitted to the application, executes the logic to assign a schema. So, the application receives data with a schema. The advantage of schema-on-application-read is that database servers usually run on more powerful server platforms and are therefore able to execute the logic faster. Especially if

Copyright © 2013 R20/Consultancy, all rights reserved.

it's a massively parallel server environment, assigning schemas, even when the values are highly complex, is fast because the logic to assign schemas is parallelized. In addition, a database server can apply filters (if relevant) so that not all the data has to be transmitted to the application. This speeds up overall performance.

Schema-On-Read Offers Flexibility to Discovery – For discovery, storing all the incoming data in its original form (schema-on-read) can be useful, because the goal of discovery is not always clear in advance. Schema-on-read allows data scientists to assign different schemas at different times to the same data without the need to restructure databases. Schema-on-read fits the flexibility requirements of data scientists.

Schema-on-read fits the flexibility requirements of data scientists.

5.2 Using SQL for Discovery

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, including those specifically designed for analytics, sometimes called *analytical database servers*. This section addresses the question whether SQL is really the right language for discovery?

Complex SQL Queries – SQL has always been a language with very strong query capabilities. In fact, in the 1970s and 1980s, SQL products were primarily deployed 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 for a database server is to run all those queries fast? The problem is that some queries are complex and hard to optimize. Let's illustrate this with a few examples.

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	Britisch Airways	9:00
:	:	:	:	:	:

Copyright © 2013 R20/Consultancy, all rights reserved.

Imagine the following 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 is a set of rows that includes the row where the DEP_ID is equal to 1. 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.

For most database servers, it's hard to process this query fast, especially if the table contains millions (or billions) 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 improves the performance.

Evidently, this is a simple example, and in reality this DEPARTURES table does not contain millions of rows. But it's easy to come up with comparable situations and queries for which millions or even billions of rows have to be accessed. For example, if a credit card company wants to see whether two charges on a credit card didn't happen too close together in a certain period of time, without any doubt massive amounts of records have to be analyzed. Or, if an organization wants to determine how many different internet sessions were started by one user, where a session is defined as a number of clicks on the website with limited time in between, again millions and millions of records may have to be accessed.

The more complex the SQL query is and the larger the data set is, the bigger the chance a 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 by customers in the same retail transaction. This question is like a market basket analysis. The corresponding SQL query is lengthy and 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
```

(continues on next page)

Copyright © 2013 R20/Consultancy, all rights reserved.

```
AND      A.REG_ID = C.REG_ID          (continuation of previous page)
AND      A.TRAN_NO = B.TRAN_NO
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 and when each individual customer buys. These tables normally contain millions and millions of rows. This means a highly complex query is executed on a very large database. It's 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 even allowed to run them at all. Such a situation definitely limits 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 influence the performance of queries, but two fundamental properties of SQL, *declarativeness* and *storage independency*, have a big impact. These 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[6, 7, 8].

Declarativeness and storage independency have always been the two properties fundamental to SQL.

When SQL was developed in the 1970s, it was supposed to be a *declarative* language. In this case, declarative means that 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 must determine how to get the requested data from the database to the user.

The second property of SQL is *storage independency*. If a system supports storage independency, it hides 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

[6] R.F. Boyce, and D.D. Chamberlin, *Using a Structured English Query Language as a Data Definition Facility*, IBM RJ 1318, December 1973.

[7] D.D. Chamberlin et al, *SEQUEL 2: A unified approach to Data Definition, Manipulation and Control*, IBM R&D, November 1976.

[8] D.D. Chamberlin, *A Summary of User Experience with the SQL Data Sublanguage*, IBM RJ 2767, March 1980.

Copyright © 2013 R20/Consultancy, all rights reserved.

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 developers.

These two language properties are 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:

- Improved Productivity: Having to write declarative code and not having to deal with the "how" implies having to write less code. This minimizes the time needed to write code compared to having to write the equivalent solution in a non-declarative language. In addition, if developers don't have to concern themselves with details related to storage and access, less code has to be designed and written.

- Improved Maintainability: For maintenance the same rules apply as for productivity: less code implies having to maintain less code. And the storage independence property makes sure that the maintenance programmer doesn't have to study the storage characteristics in order to make the necessary changes.

- Improved 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 (see Figure 2), indicated in his seminal paper[9], written in response to his receipt of the ACM Turing award, that his goal for developing the relational model was *data independence* (which relates to the term 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."*

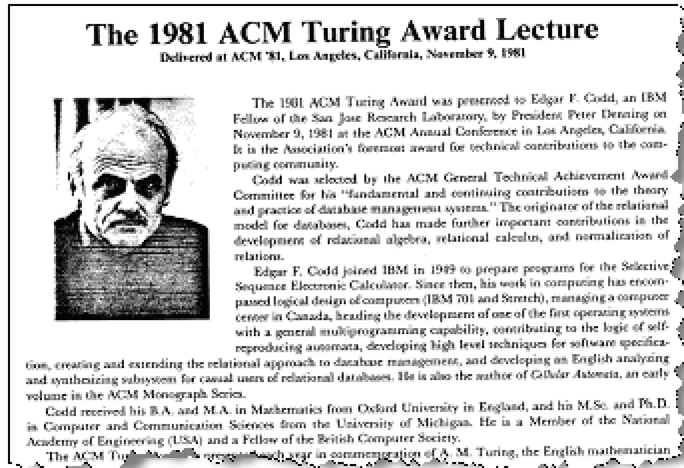
Why Are SQL Queries Sometimes Slow? – Why can declarativeness and storage independency have a negative impact on performance? Because of these properties, a SQL database server has to transform a query into an *access plan* that describes in detail how the data should be accessed, joined, grouped, filtered, and so on. Such an 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, for some queries it remains hard to come up with an efficient access plan. This is one of the main reasons why performance is not always perfect.

---

[9] E. F. Codd, *Relational Database: A Practical Foundation for Productivity*, Turing Award Lecture in Communications of the ACM, Volume 25, Number 2, February 1982.

Copyright © 2013 R20/Consultancy, all rights reserved.



The 1981 ACM Turing Award Lecture

Delivered at ACM '81, Los Angeles, California, November 9, 1981

The 1981 ACM Turing Award was presented to Edgar F. Codd, an IBM Fellow of the San Jose Research Laboratory, by President Peter Denning on November 9, 1981 at the ACM Annual Conference in Los Angeles, California. It is the Association's foremost award for technical contributions to the computing community.

Codd was selected by the ACM General Technical Achievement Award Committee for his "fundamental and continuing contributions to the theory and practice of database management systems." The originator of the relational model for databases, Codd has made further important contributions in the development of relational algebra, relational calculus, and normalization of relations.

Edgar F. Codd joined IBM in 1949 to prepare programs for the Selective Sequence Electronic Calculator. Since then, his work in computing has encompassed logical design of computers (IBM 701 and Stretch), managing a computer center in Canada, heading the development of one of the first operating systems with a general multiprogramming capability, contributing to the logic of self-reproducing automata, developing high level techniques for software specification, creating and extending the relational approach to database management, and developing an English analyzing and synthesizing subsystem for casual users of relational databases. He is also the author of *Cellular Automata*, an early volume in the ACM Monograph Series.

Codd received his B.A. and M.A. in Mathematics from Oxford University in England, and his M.Sc. and Ph.D. in Computer and Communication Sciences from the University of Michigan. He is a Member of the National Academy of Engineering (USA) and a Fellow of the British Computer Society.

The ACM Tu[illegible] each year in commemoration of A. M. Turing, the English mathematician

Figure 2 *Edgar F. Codd's ACM Turing Award lecture.*

SQL Improves Discovery Productivity – The declarativeness and storage independency of SQL are beneficial to productivity, maintenance, and flexibility. Even users with limited knowledge of databases can write queries. The answer to the question raised at the start of this section whether SQL is really the right language for discovery, is yes. SQL is a highly recommended language for data scientists to be used for preparing and analyzing data, because of its high productivity, high flexibility, and low maintenance. A drawback of SQL database servers may be that the queries that are hard to optimize may perform poorly.

SQL is a highly recommended language for data scientists to be used for preparing and analyzing data.

5.3 Functions in SQL

A feature of SQL that enriches its analytical capabilities is the *function*. Functions are not new to SQL. In fact, the first versions of SQL already supported them, although those first functions were simple ones, such as truncate a string and calculate the square root. Through the years most SQL implementations have been extended with more advanced functions, some of them designed specifically for analytics.

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 SQL developers who write SQL statements that invoke the functions. To keep SQL's declarative and storage independent properties intact, it's important that SQL developers don't need to concern themselves with how functions work, in which language they have been coded, and so on. That should only be relevant to the function developer.

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

Built-in Functions and 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*. SQL developers have no idea in which language

Copyright © 2013 R20/Consultancy, all rights reserved.

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. The developer writing the SQL statement doesn't see the difference between those two types of functions. For him the SQL code is still declarative, while the developer, who wrote the function `BANK_HOLIDAY`, may have used a non-declarative language, such as Java and C++.

Scalar Functions and Table Functions – The second way of classifying functions is by the result they return. Functions exist 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 in which each row consists of the same number of values. The former ones are called *scalar functions*, and the latter *table functions*.

The functions `TRUNCATE` and `BANK_HOLIDAY` are both examples of scalar functions. Other examples of scalar functions are change a dollar value into a euro value, and subtract 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 (for example, 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, and External Functions – The third way of classifying functions is based on the language used to code them:

- 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, such as `INSERT`, `DELETE`, and `UPDATE`.

- Procedural functions: The bodies of these functions are written with declarative 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. Non-declarative statements are supported by many SQL database servers,

Copyright © 2013 R20/Consultancy, all rights reserved.

including DB2, Oracle, Sybase, and Teradata. 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 may 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: a *simple function* doesn't, whereas 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, making it a complex function.

Functions Enrich SQL for Discovery – When a SQL system is used as discovery platform, it's important that it supports many built-in analytical functions and that it allows that UDFs are developed to enrich the analytical functionality. This way, SQL stays declarative and storage independent, improving productivity and maintenance, and making it easy to write SQL queries that invoke advanced analytical functions.

When a SQL system is used as discovery platform, it should support many built-in analytical functions and the development of UDFs.

Note: Extending SQL with functionality by adding functions is not new. For example, functions have been added to support manipulating and querying XML documents. Some SQL systems even offer functions to extend SQL statements with XPath and XQuery expressions. In this case, SQL operates as a host language for those other languages. Others have extended SQL by adding functions for data mining algorithms.

5.4 Parallelization of SQL Queries and SQL Functions

As indicated, performance and scalability are important to discovery. A discovery platform must be able to run even complex forms of analysis fast, and it should be able to do this on massive amounts of data. This section describes internal architectures of database servers and how they use parallelization to improve performance.

A Parallel Database Server – To speed up query performance, most database servers can exploit multi-processor hardware by distributing the query processing over as many processors as possible. This is called *parallelization of queries*. This section explains why parallelizing queries can improve performance, but that some queries are hard to parallelize. We begin by introducing some terminology.

In order to distribute query processing over multiple processors, the architecture of many database servers is distributed. Figure 3 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

Copyright © 2013 R20/Consultancy, all rights reserved.

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.

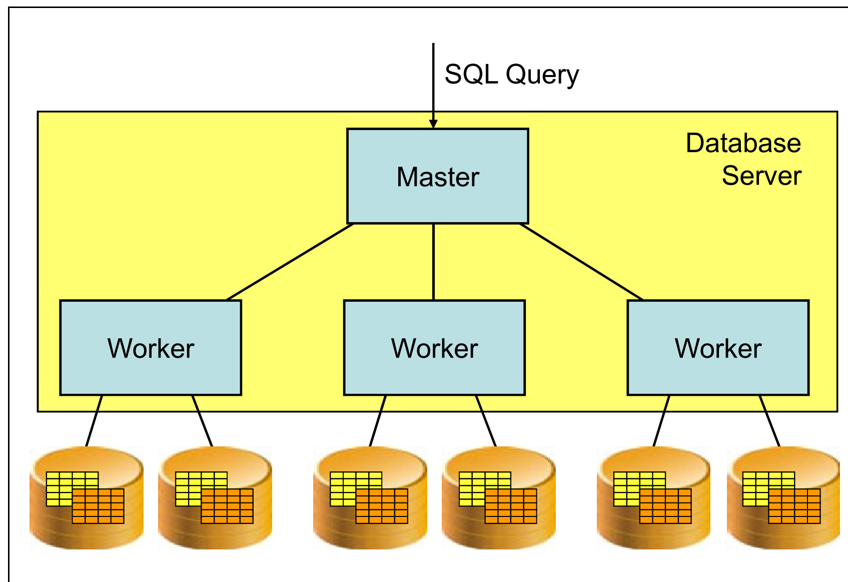


Figure 3 *Typical architecture for parallel database servers.*

When an application sends a query to the database server, it's first transmitted to the Master; that's where all the processing starts. The Master breaks a query in a number of smaller queries depending on which tables are accessed, on which nodes those tables are located, and how the table rows are partitioned. 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, if needed it does some extra processing, 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 broken into even smaller snippets and those are shipped to lower level Workers. These Workers return 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.

The main goal of this architecture is to let the Workers do as much as possible of the query processing in parallel, and to let the Master(s) do as little as possible, so that the Master doesn't become the 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 processors. Another form, called *intra-query parallelism*, is where the processing required for one particular query is distributed over multiple processors. The first form improves the workload, whereas the second improves the response time of individual queries.

Two sub-forms exist of the second form: *inter-operation parallelism* and *intra-operation parallelism*. Each query is 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

Copyright © 2013 R20/Consultancy, all rights reserved.

operations (belonging to one query) is distributed over multiple processors. This can definitely improve the query response time. However, if one of the operations involves a scan of millions of rows, and that scan is not parallelized, that one operation can still take minutes to complete, therefore slowing down the processing of the entire query.

With intra-operation parallelism the processing of an operation, such as a scan, is distributed over multiple nodes. Intra-operation parallelism requires that tables are partitioned. If the partitions of a table 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 that 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 that this whole notion of parallelization is hidden for SQL developers. This is an aspect of the storage independency property of SQL. Developers don't and shouldn't have to indicate how to parallelize queries. Else, it 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, which operations can be executed in parallel by the Workers?

A few examples are used to show how complex parallelization can be. We start with the following simple query:

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

For most database servers this query is easy to parallelize its execution. The Master can send the entire query as a snippet to each Worker. Each of the Workers only returns those rows (and a few columns) for which the condition `PRICE > 100` is true. The effect is that the processing of the entire query is parallelized. The only thing left for the Master to do is to combine the results from the Workers using a simple union operation.

But what if a condition contains complex calculations, or a subquery, or if it invokes a complex UDF? Hopefully, the database server is smart enough to include all these operations inside the query snippets to be sure that the Workers return 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 are send back to the Master, the Master has to perform all the extra processing serially.

When the Master has to do a lot of processing, it becomes the bottleneck of the entire system. This has a negative impact on the scalability of the system. Adding more Workers to the architecture doesn't solve that problem; as reflected in Figure 4. The overall performance suffers, because too much query processing is not executed in parallel. Note that this does not apply to Teradata database servers.

Copyright © 2013 R20/Consultancy, all rights reserved.

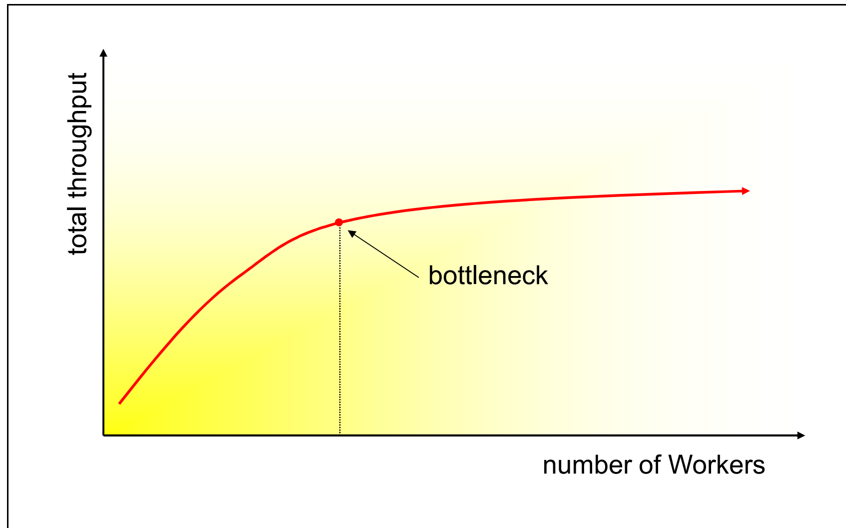


Figure 4 *When the Master must do too much processing, it becomes a bottleneck that severely limits the scalability of the entire system.*

Most analytical queries contain group-by operations. 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 even impossible to perform group-by operations in parallel if the records in the SALES_RECORDS table have not been partitioned on the column REGION_ID. In such a situation, they probably send the following query snippet to each of the Workers:

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

This snippet doesn't contain any group-by operation. The consequence is that 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, therefore it's much more efficient if the group-by operations are processed by the Workers, because a much smaller set of rows is returned.

The first query in Section 5.2 is the next example. What should an optimizer do with the correlated subquery? What we don't want is that the subquery is executed for each row in each table, because it's inefficient. Plus, it means that for each row the subquery is send to the Master for processing, and this happens over and over again. In addition, if the table accessed in the subquery is large, it will be very slow. Conclusion, it's difficult to come up with a fully parallelized access plan for this query.

Time-series based queries are also hard to parallelize. In these queries, rows are selected based on a row's values and the values of the previous or next row. For example, imagine rows have to be selected where the value of a column is greater than the value of the same column in the previous row. Most database servers process this query by letting the Workers return all the

Copyright © 2013 R20/Consultancy, all rights reserved.

rows to the Master and by letting the row selection process be handled by the Master itself. Evidently, this is inefficient.

Parallelizing Function Processing – Parallelizing simple scalar functions is not that difficult for most database servers. They push the processing of those functions down to the Workers. For example, if the comparison `function(column) = value` is included in a condition, its processing can be distributed to all the Workers, and each Worker only returns those rows to the Master that adhere to this condition. This assumes that all the data needed to evaluate the condition is available in the row being studied.

Parallelizing complex functions is much more difficult. Imagine that a scalar function called `NEXT_FLIGHT` has been developed that simplifies the long query in Section 5.2. This function determines whether another flight departs to the same city within one hour. This function has three input parameters and returns a 1 if another flight is available, and 0 if there isn't. A rewrite of the query but now using the function looks 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. In which 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 others row than by using SQL statements, and this is regardless of the language in which the function is coded. If these extra queries are executed, they are sent to the Master that has to determine how to execute them. If this is done for each row, an avalanche of queries is returned to the Master. Evidently, this is a very time consuming process.

Some functions are *deterministic*. A deterministic function returns the same value every time it's executed. When such a function is used in a query, the Master can execute it first and substitute the function call in the condition by its return value. The Master can then send the query snippet with the substitution to the Workers. In this case, the Workers won't invoke extra queries and the function processing can easily be parallelized.

However, in the above SQL example, the function is not deterministic. For each individual row the function must be executed, thus for each row, a query has to be executed. So, instead of processing one query, the database server must 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.

Parallelizing SQL Queries and Schema-On-Read – Section 5.1 describes the different forms of handling schemas in databases. If schema-on-database-read is used, somehow the SQL queries have to be extended with the logic to determine the schema for the schema-less values before the data is transmitted to the applications.

SQL functions can be developed to handle the process of assigning schemas to values. If this logic is straightforward enough to be implemented as simple scalar functions, the Workers can process it in parallel. However, when complex functionality is involved, it may have to be

Copyright © 2013 R20/Consultancy, all rights reserved.

implemented as complex table functions. This increases the chance that the processing of these functions is not handled by the Workers, but must be executed by the Master. This influences the scalability and performance of the system negatively. In this case, it's better to change from schema-on-database-read to schema-on-application-read.

Summary – Performance and scalability are crucial for a discovery platform. Therefore, it's important that the platform runs complex queries fast. Fast means that all the query processing, including all the complex analytical operations and the complex schema-on-database-read logic, is executed in parallel. The more logic is processed by a central component, the slower the performance will be. Therefore, it's important that the database server offers features to parallelize complex queries including the function processing.

5.5 Hadoop and MapReduce in a Nutshell

The focus of the previous sections is on SQL-based systems, this one discusses *NoSQL systems*. As indicated in Section 5.1, a new generation of NoSQL systems is introduced for developing big data systems. NoSQL systems can be seen as potential technologies for developing discovery platforms. This section describes some key NoSQL technologies, including MapReduce, Hadoop, HDFS, and SQL-fication of Hadoop.

Introduction to 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 technology that is based on MapReduce. MapReduce is used for offline batch processing to build the search indexes. Then, when someone searches for a term with Google, the lookup is done with those indexes. So, even though we may not be aware 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*[10]. In this article they introduced MapReduce, a programming model for processing requests on large datasets in which 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 patented[11] since January 2010. Google and other companies use this programming model to maximize parallelization and improve response times.

We emphasize that MapReduce is a *programming model* and not a programming language. It's a style of solving a specific problem. In principle, a developer can use any programming language for implementing a MapReduce-based solution.

MapReduce is a programming mode and not a programming language.

[10] 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.

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

Copyright © 2013 R20/Consultancy, all rights reserved.

The words *Map* and *Reduce* stand for the two types of operations in which requests are split up. Map operations are good for filtering records and for applying logic to individual values, such as string and mathematical operations. Reduce operations are intended for combining records with comparable values into one record. When compared with SQL, Map operations combine the functionality of the `SELECT` and `WHERE` clauses of SQL's `SELECT` statement, and Reduce operations are like the `GROUP BY` clause.

The implementation of MapReduce determines how these functions are really processed. Later in this section, we describe how MapReduce has been implemented in Hadoop.

Introduction to Hadoop – *Hadoop* is a software framework designed for supporting data-intensive applications. It's for those applications in which a continuous stream of new, incoming data has to be stored and managed, and where all that data has to be analyzed periodically. Examples of such applications are click stream applications that generate enormous amounts of records, and sensor-driven application (RFID-based) that require a continuous stream of measurements to be stored. Some of these applications literally generate thousands of records per second. All this data needs to be stored for future use, leading to a massive amount of data storage. Hadoop has been designed to support this type of application. In other words, it has been designed for the world of big data.

As indicated, Hadoop has the capacity to analyze large portions of all that data. Because of this feature, Hadoop is often positioned as a potential discovery platform, and is therefore described in this section.

The Modules of Hadoop – Hadoop consists of a set of modules. We briefly introduce the core modules here. For a more extensive description, we refer to Tom White's book[12] on Hadoop.

- HDFS: The foundation of Hadoop is formed by *HDFS* (Hadoop Distributed File System). This module is responsible for storing and retrieving data. It's designed and optimized to deal with large amounts of incoming data per second and for managing enormous amounts of data up to the petabytes. The key aspect of HDFS is that it can distribute its data over a large number of disks and can exploit an MPP architecture. HDFS supports a well-designed programming interface that can be used by any application. HDFS is the only mandatory module of Hadoop, the others are all optional.

- MapReduce: The module called *MapReduce,* as the name suggests, implements Google's MapReduce programming model. This component allows that data inserts and data queries are efficiently distributed over hundreds of nodes. Important to note is that the programming interface of Hadoop's MapReduce is very technical and requires a deep understanding of the internal workings. It does not support storage independency.

- HBase: The *HBase* module is designed for applications that need random, real-time, read/write access to data. It operates on top of HDFS.

- Hive: 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 some are missing, such as subqueries in the `SELECT` and `FROM` clause, and the `HAVING` and

[12] White, Tom, *Hadoop, The Definitive Guide*, O'Reilly Media, 2012, third edition.

Copyright © 2013 R20/Consultancy, all rights reserved.

LIMIT clause. Internally, Hive translates the SQL statements to MapReduce batch jobs. By doing this, the processing is parallelized.

- Pig: Next to Hive, developers can also use *Pig* for querying the data. The language supported by this module is called Pig Latin and is more technical 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.

Processing MapReduce in Hadoop – Applications using Hadoop's MapReduce invoke logic by executing a set of Map and Reduce steps. During a 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). In most cases, a request 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 that a strong resemblance exists here with breaking a SQL query into multiple scan operations and distributing them over as many nodes as possible.

Let's illustrate the 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 that the input parameter of this function is MIN_AMOUNT. This means that 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 is a set of records indicating the total amount of sales for each store. In the Map step, this call is distributed over as many nodes as possible. The resulting records are stored in intermediate files. It's up to the developer to determines where these files are stored. It's important that these files are stored in such a way that it makes parallelization of the next step easy.

A Reduce step groups records with similar values. In fact, its operation resembles the GROUP BY clause in SQL. This step is called Reduce, because only a reduced number of records is returned. The result of the Reduce step is also a set of files.

A MapReduce program is not limited to having one Map and one Reduce step, many Maps and Reduces are allowed. At the end of the last step, the application reads all the intermediate results.

As with parallel database servers, the goal of Hadoop's MapReduce is to minimize the amount of data returned to the next step. The logic of the various steps is determined by the developers. They can use any programming language construct to make these steps as efficient as possible, they determine the order in which logic is processed, how and where processing takes place, and where intermediate results are stored. In other words, the code inside the steps is non-declarative and storage dependent. The big advantage is that the developer has full control over the processing strategy. The disadvantage is that the performance is determined mainly by the quality of the developers.

Copyright © 2013 R20/Consultancy, all rights reserved.

The Batch-Oriented Nature of MapReduce – MapReduce programs are executed as batch jobs, which are scheduled and distributed over many nodes. In the background the process of starting and running all those jobs is monitored and managed. Distributing and managing all these jobs, requires additional processing. However, considering the amount of data analyzed, this overhead of additional processing is probably acceptable for a non-interactive analytical environment. Note that some database servers would not even be able to query so much data, so this extra time for management is the penalty paid for being able to analyze these huge amounts of data with an adequate performance.

The SQL-fication of Hadoop – Lately, more and more modules have been released that offer SQL interfaces to Hadoop. Cloudera has released Impala, HortonWorks has Stinger, and MapR technologies will release Drill. In addition, data virtualization vendors, such as Composite Software, Denodo Technologies, and Informatica, have also made their products available for Hadoop. SQL interfaces are becoming available for other NoSQL systems as well, such as CQL for Cassandra. Furthermore, some SQL database servers support access to Hadoop. Aster Database is one of them; see Section 6.3.

There is a growing demand for *SQL-fication* of Hadoop. Organizations want to have an interface to Hadoop data that is easier to use than the HDFS or MapReduce interfaces.

There is a growing demand for SQL-fication of Hadoop.

5.6 The Marriage of SQL and MapReduce: SQL-MapReduce

Some database vendors have combined SQL with MapReduce by creating a SQL database server based on a MapReduce architecture. *Teradata Aster Database* (formerly called Aster Data *n*Cluster) is one of these products that have implemented SQL-MapReduce to make SQL more suitable for analytics.

Aster Database is part of the *Teradata Aster Discovery Platform,* which also includes the *Teradata Aster Discovery Portfolio*. The first version of Teradata Aster Database was released in 2006, and the first production deployment was in 2007. The current version 5.10 was released in the first half of 2013.

This section describes how in Aster Database the MapReduce programming model has been implemented to exploit parallel hardware and to fully parallelize query processing and therefore make analytics possible even on commodity hardware. Note that other vendors have implemented MapReduce as well, but all these implementations differ.

Teradata Aster Database's SQL-MapReduce – From the SQL developer's perspective, the entire MapReduce framework is implemented as a set of *external table functions* that can be invoked from SQL. This means that report developers won't have to learn a new language. They only have to familiarize themselves with the parameters and the way in which these table functions must be invoked. Because the MapReduce table functions are according to E. F. Codd's rules for the relational model, SQL remains a declarative and storage independent language.

Copyright © 2013 R20/Consultancy, all rights reserved.

The next example shows what SQL-MapReduce means to developers. We use the table plus the complex query from Section 5.2. 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 much 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 that must 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 a row exists 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 to 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, and more importantly, Teradata Aster Database 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 part (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 5.2 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 version of the query is much simpler, and it's obvious that it's easier to improve the performance of this query. The function `BASKET_GENERATOR` is designed specifically 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.

To summarize, in Teradata Aster Database when an analysis technique can be written as a MapReduce table function, its processing is fully parallelized. This even applies to complex forms of analysis (note that this is not true for table functions in classic SQL systems). Overall, this is a big advantage for data scientists.

Copyright © 2013 R20/Consultancy, all rights reserved.

Schema-on-Read Functions – Because Teradata Aster Database is able to distribute the processing of MapReduce functions over many processors, it can support high-performance schema-on-read. Imagine that schema-less, weblog messages comparable to the one in Section 5.1 are stored in a table. In this case, MapReduce functions can assign schemas to these values and return the data in a set of structured columns. The processing of these functions may be intensive, but because the processing is parallelized, it doesn't hurt the query performance.

> Teradata Aster Database supports high-performance schema-on-read.

Teradata Aster Database and Hadoop – Teradata Aster Database comes with a built-in capability called SQL-H™ for accessing data stored in HDFS. This capability provides metadata integration with Hadoop and makes it completely transparent for data scientists whether data is coming from Aster's own database or from HDFS. The data scientists don't have to deal with the HDFS technical details and the low-level interfaces nor with writing efficient Hadoop MapReduce programs. They only have to invoke the functions to treat Hadoop data in the same way as Aster data.

> SQL-H makes it completely transparent for data scientists whether data is stored in Aster's own database or in HDFS.

The functions that access HDFS don't use Hadoop's MapReduce, but use Aster's own SQL-H technology to extract data from HDFS in a parallel way. If the query contains filters that can be pushed down to HDFS, Aster will do so. The effect is that less data is returned to Aster and this speeds up processing. The Hadoop data that is retrieved, is kept in memory by Aster.

To speed up the performance, the result of the query on HDFS can also be made persistent in an Aster table. This can be useful, for example, when a result has to be reused multiple times.

Rich Set of Built-in Analytical Functions – The Teradata Aster Discovery Platform features the Teradata Aster Discovery Portfolio which offers an extensive set of pre-built functions to support each step of the discovery process from data acquisition to data analysis. These functions provide capabilities for statistical analysis, relational analysis, path analysis, affinity analysis, pattern matching, graph analysis, visualization, and text analysis (see Appendix A at the end of this whitepaper for a list of functions currently supported). These functions can be mixed and matched as in the following example:

```
SELECT * FROM nPathViz (
   ON SELECT * FROM nPath (
      ON SELECT * FROM SESSIONIZE (
         ON SELECT * FROM LOAD_FROM_TD_HADOOP
...
```

Here four MapReduce functions supplied with the Teradata Aster database have been nested. First, the function `LOAD_FROM_TD_HADOOP` is used to extract data from Hadoop. Next, the data is sessionized. This function prepares the complex multi-structured weblog data for analysis. Next, the function nPath is used to identify paths in the data. This is the second form of analysis deployed, and finally the function nPathViz is invoked to visualize the data. The result of this query is shown in Figure 5.

Copyright © 2013 R20/Consultancy, all rights reserved.

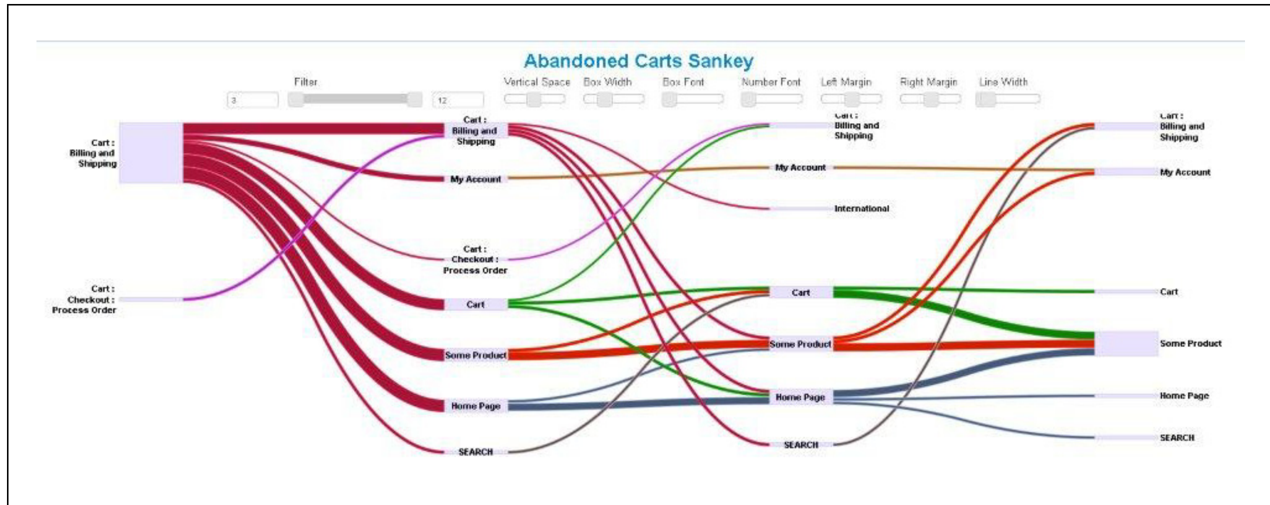


Figure 5 *The visual result of deploying multiple analysis techniques, one after another.*

Besides the visualization presented in Figure 5, Teradata Aster Discovery Portfolio provides a rich set of visualization functions, including the one in Figure 6.

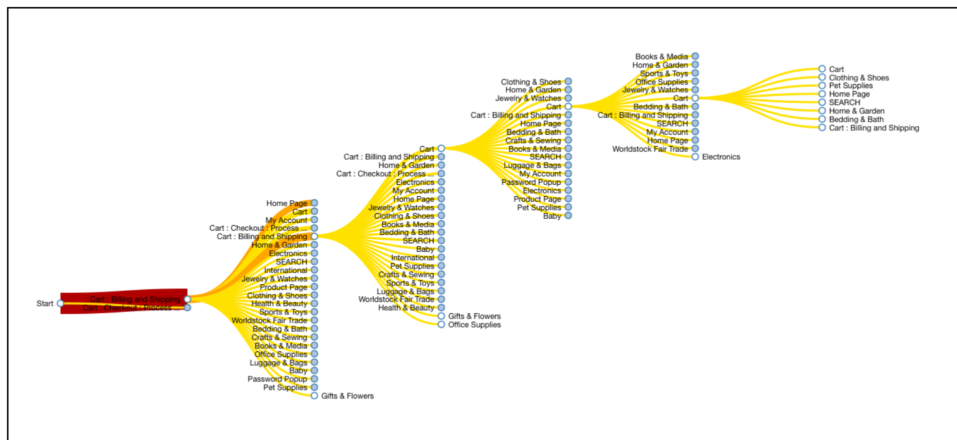


Figure 6 *Data visualization example of cart abandonment where a drilldown has been deployed by product category.*

6 Implementing a Discovery Platform

Data scientists can select from different solutions for implementing a discovery platform. In this section the following five are described:

1. Classic SQL system
2. Advanced Reporting Platform
3. SQL-MapReduce System
4. Hadoop with MapReduce
5. Hadoop with SQL interface

In the coming sections, these solutions are described in detail, and Chapter 7 contains an overall high-level comparison.

Copyright © 2013 R20/Consultancy, all rights reserved.

6.1 Solution 1: Classic SQL System

From many perspectives, the preferred solution is that data scientists use a classic SQL system that the organization has already installed. The advantages are:

- SQL is a high-level development language that's likely known to most data scientists.
- Many organizations already use a SQL system, so DBA's know how to manage, tune, and optimize it.
- A large set of reporting and analytical tools is able to exploit data stored in classic SQL systems.
- SQL systems are very much suited to support interactive analysis where data scientists constantly execute new SQL queries.
- The optimizers of most SQL systems are very mature and are capable of coming up with efficient access plans for most queries.

Although an attractive option, this solution does have disadvantages:

- Each SQL system supports some simple statistical functions, and some even offer some data sampling and data mining functionality, but no advanced statistical or visualization functions are available. The analytical capabilities are limited and therefore additional analytical tools are needed to fill this functionality gap.
- Most SQL systems are not data integration platforms. This means that when data stored in the SQL system has to be integrated with data coming from other systems, the data scientist has to organize this himself. This probably requires extra tools and redundant storage of data. If those SQL systems can integrate their data with data from external sources, these external sources are usually limited to SQL-like systems. Non-SQL systems are not supported.
- Section 5.4 describes why most classic SQL systems can't parallelize the more complex analytical operations. This does not only severely limit the query performance of the platform, but the data scalability as well.
- Classic SQL systems are designed and optimized for supporting schema-on-write, so when schema-less values, such as the ones shown in Section 5.1, are stored, extra application logic must be developed to unravel the structure hidden in these values. In many situations, this is usually done using schema-on-application-read, thus lowering the performance and scalability of the system.

6.2 Solution 2: Advanced Reporting and Analytical Platform

Many tools exist that offer powerful reporting and analytical capabilities. Some come with built-in in-memory data store solutions, and others with their own disk-oriented storage solutions. Most of them have been designed to let database servers handle the storage and processing of data.

Copyright © 2013 R20/Consultancy, all rights reserved.

The advantages of using a reporting and analytical tool as a discovery platform are:

- Once the data is loaded into internal memory, these reporting tools definitely run queries fast. Regardless of what the data scientists want to do with the data, the reply is almost instantaneous. This fits the interactive style of the discovery process well.
- Most of these tools support easy-to-use graphical, user-friendly interfaces which improves productivity.

The disadvantages of this solution are:

- Data scalability is limited. It's true that today machines can have much more internal memory than a few years ago, still, the amount of data to be analyzed in a big data environment is too much to load into internal memory. In other words, data scalability is not their strongest point.
- No reporting or analytical tool supports all the imaginable types of analysis. The consequence is that data scientists need to switch between multiple tools (each offering its own strengths). Mixing and matching analysis results between tools is not evident.
- More and more of these tools come with built-in data integration tools. However, the way these modules work is that real data integration takes place after all the data has been loaded into memory, severely limiting data scalability and performance.
- Most of these tools don't know how to process schema-less values. Schema-on-write is the preferred approach.

6.3 Solution 3: SQL-MapReduce System

Section 5.6 describes the SQL-MapReduce solution offered by Teradata Aster Database. Teradata Aster Database's internal architecture offers many advantages for implementing a discovery platform:

- This platform offers data scalability and high-speed analysis because of its MapReduce-based architecture, which is responsible for parallelizing most of the query and function processing.
- Aster Database comes with a large set of pre-built analytical functions (see Appendix A) that can be mixed and matched.
- Data scientists work with a familiar language: SQL. They don't have to learn a complex, low-level technical language.
- Most reporting and analytical tools support SQL and are therefore able to access the data stored in the Aster Database and also in Hadoop. This allows the data scientists to deploy any reporting and analytical tool they want.
- When data has to be integrated with Hadoop data, it can be accessed in a transparent way. There is no need for data scientists to learn how to work with Hadoop.
- With the MapReduce functions, the execution of complex schema-on-database-read logic can be parallelized.
- Aster Database comes with an integrated development environment for developing and testing MapReduce functions and SQL statements.
- The optimizer in Aster Database is mature and can handle multi-table joins efficiently, including queries that access Hadoop data.

Copyright © 2013 R20/Consultancy, all rights reserved.

- Like most SQL systems, Aster is well-suited for interactive analysis.

The disadvantages of this solution are:

- Although the Teradata Aster Discovery Platform supports several powerful interactive graph forms to visualize data, such as affinity graphs and path graphs, other tools may be needed to extend the graphical capabilities.
- Data retrieved from Hadoop is loaded into memory. The amount of data loaded can be too much for Aster. Evidently, this is dependent on the hardware configuration. If this is the case, the Hadoop data can be saved as a table in Aster.
- The Aster Database is designed for discovery; it's its claim to fame. The platform is not ideal for more classic forms of reporting. For this purpose a separate environment may be needed.
- Some developers must learn how to write MapReduce functions. However, 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 it's usually a small group of specialists that develop functions, not the entire 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.
- 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.

6.4 Solution 4: Hadoop with MapReduce

The Hadoop modules HDFS and MapReduce together form a potential discovery platform. The advantages are:

- Developers can write MapReduce programs that are completely parallelized when executed. This makes this platform highly scalable.
- MapReduce functions can be developed for performing many different forms of analysis.
- MapReduce functions can be developed for performing schema-on-database-read operations on schema-less values.
- Developers, who know all the technical characteristics of Hadoop, can fully exploit the potential power of Hadoop.

The disadvantages of this solution are:

- Developing in MapReduce is cumbersome, because it supports a low-level interface. It requires considerable technical skills. This requires the data scientists to develop in Java, or outsource this work to external developers.
- MapReduce code is neither declarative nor storage independent, which has a negative impact on productivity and maintenance; see Section 5.2.
- Because MapReduce is a batch oriented environment and programming is done in a low-level interface, it's not so much suited for interactive analysis.

Copyright © 2013 R20/Consultancy, all rights reserved.

- If MapReduce is used, no data stored outside HDFS can be accessed. If data from other data sources has to be analyzed, that data has to be loaded into HDFS first. This requires separate tools, and causes redundant data storage.
- Hadoop doesn't come with pre-built analytical functions. These functions must be developed by hand, a library with such modules has to be acquired, or a separate product must be acquired.
- Programming joins that are processed in parallel is complex on MapReduce.
- Hadoop is not ideal for more classic forms of reporting. For this purpose a separate environment may be needed.

6.5 Solution 5: Hadoop with a SQL Interface

As indicated in Section 5.5, SQL interfaces, such as HiveQL, Cloudera Impala, MapR Drill, and HortonWorks Stinger, exist for accessing data stored in HDFS. While HiveQL accesses data in HDFS via MapReduce, the other interfaces have their own engines to access HDFS data—they bypass MapReduce.

All these interfaces have implemented a SQL optimizer that translates SQL queries into programs for accessing HDFS data in parallel. The challenge of these optimizers is to come up with access plans in which all the processing is parallelized. If some processing is not done in parallel, the interface has to do that processing itself, which lowers query performance.

The advantages of deploying a SQL interface on HDFS are:

- A SQL interface is a higher-level interface than that of MapReduce. This improves productivity and flexibility, which is beneficial for the interactive character of discovery.
- Because these engines run on HDFS, the solution offers a high level of data scalability.
- The products that bypass MapReduce are not batch-oriented and are therefore more suited for interactive environments such as discovery.
- Because of their SQL interfaces, many popular reporting and analytical tools can be used to access the data in HDFS.

The disadvantages of this solution are:

- Most Hadoop-based SQL interfaces can't access data stored outside HDFS. As with the previous solution, this requires that data is copied into HDFS first, which takes time, increases storage costs, and slows down all the queries because of the increased size of data. It has to be noted that MapR claims that the first version of Drill supports access to other data stores than HDFS.
- These SQL interfaces do not come with pre-built analytical functions making them more suitable for reporting than for analytics. Note that such functions can be developed.
- When analytical functions have to be developed, a low-level language must be used. These functions cannot be developed with SQL itself.
- The optimizers in most SQL systems needed many years to mature to a level that they could come up with efficient access plans for most queries, including multi-table joins. The optimizers of the new SQL interfaces are young. The question is how much time they

Copyright © 2013 R20/Consultancy, all rights reserved.

need to mature? Evidently, they don't need as many years as the SQL systems, because they can learn from the older systems.

- Having a SQL interface doesn't make a solution suitable for data scientists. They need tools with user-friendly and intuitive interfaces. Most of the new SQL interfaces don't support integrated development environments. Although this problem may be solved shortly by linking up with one of the many existing IDEs for SQL.
- DBAs who are familiar with managing classic SQL databases have to familiarize themselves with the Hadoop environment. For example, they have to study how data is backed up and recovered, and how data should be distributed efficiently.
- Not all these SQL interfaces support schema-on-read. For some, data has to be stored in a relational way: schema-on-write.
- Most of the new SQL interfaces have only implemented a subset of the ANSI SQL standard.

7 Comparison of Five Solutions for Implementing a Discovery Platform

In this chapter the five solutions described in the previous chapter are compared. Table 1 shows how well the five solutions meet the requirements of a discovery platform as listed in Section 4.

Requirements for a Discovery Platform	Classic SQL Systems	Advanced Reporting and Analytical Platform	SQL and MapReduce	Hadoop with MapReduce	Hadoop with SQL Interface
Data scalability	Medium	No	Yes	Yes	Yes
Multi data store access	No	Yes	Yes	No	No
Complex, schema-less value analysis	No	No	Yes	Yes	Yes
Data preparation techniques	No	Some	Yes	Yes	Yes
Multiple analysis techniques	No	Yes	Yes	No	No
Multiple analysis tools	Yes	No	Yes	No	Yes
Interactive analysis	Yes	Yes	Yes	No	Yes/No
High-speed analysis	Medium	Yes	Yes	Yes	Yes
High development language	Yes	Yes	Yes	No	Yes

Table 1 *Comparison of the five solutions based on the requirements for a discovery platform.*

Copyright © 2013 R20/Consultancy, all rights reserved.

Table 2 contains a more technical comparison of the five options.

Technical Characteristics	Classic SQL Systems	Advanced Reporting and Analytical Platform	SQL and MapReduce	Hadoop with MapReduce	Hadoop with SQL Interface
Online query processing	Yes	Yes	Yes	No	Yes
Declarative and storage independent API	Yes	Yes	Yes	No	Partially
Support for schema-on-database-read	No	No	Yes	Yes	Yes
Pre-built analytical functions	No	Yes	Yes	No	No
Comes with integrated development environment (IDE)	Yes	Yes	Yes	No	No
Efficient multi-table join processing	Yes	No	Yes	No	No
Heterogeneous data access	No	Yes	Yes	No	No
Not in-memory copy of data	Yes	No	Yes	Yes	Yes
Familiarity of interface to BI developers	High	High	High	Low	High
Accessible by most reporting and analytical tools	Yes	No	Yes	No	Yes
Who or what acts as optimizer?	SQL system	Underlying data store	Aster Database	Developer	SQL interface
Difficulty of developing user-defined functions	Difficult	Easy	Easy	Complex	Complex
Also suitable for classic reporting	Yes	Yes	No	No	Yes/No
Database administration concepts familiar to data warehouse administrators	Yes	Based on the underlying data store	Yes	No	No

Table 2 *Technical comparison of Teradata's Aster Database and the various interfaces supported by Apache's Hadoop.*

The two Hadoop options 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 and being able to analyze that data fast. Hadoop is currently not ideal for interactive analytics where users can go back and forth between queries and results, and expect instantaneous results.

In this comparison, the SQL-MapReduce scores quite favorably. The strength of SQL-MapReduce is that it can manage large amounts of stored data, uses a familiar and high-level development language, and supports all forms of analytics including those forms where users interactively analyze data. In addition, Teradata Aster Database's SQL-MapReduce

Copyright © 2013 R20/Consultancy, all rights reserved.

implementation comes with a suite of 70+ pre-packaged analytic modules and an integrated development environment to help analysts and data scientists be productive more quickly.

Furthermore, because Teradata Aster Database supports standard SQL, interactive analysis, and multi store data access, business analysts can use almost any type of tool to create reports or analyze the data. In other words, this platform is not only suitable as a discovery platform, it can also be used for more traditional forms of reporting and analysis, even when the data can be classified as big data and contains schema-less values.

8 Technical Advantages of SQL-MapReduce

This chapter lists some of the technical advantages of Aster Database's SQL-MapReduce implementation when deployed as a discovery platform.

Parallelization of Complex Operations – Operations that are hard to parallelize by most database servers, such as joins, group-by's, 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 – Data scientists 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 – The MapReduce functions in Aster Database 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 that 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.

Big Data Access – Aster Database comes with a built-in capability called SQL-H™ to access data in HDFS. Developers see no difference between accessing data in an Aster database or data in HDFS, this is all transparent. In addition, all the analytical functions that can be deployed on data in an Aster database, can be deployed in the same way as on data in HDFS.

Schema-on-Read – Although Aster is a SQL database, it works efficiently with complex values. MapReduce functions can be developed that transform the complex values in simple values when the data is extracted from disk and before it's passed to the applications. Because the execution of these functions is parallelized, schema-on-database-read is fast.

Copyright © 2013 R20/Consultancy, all rights reserved.

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 right before it's executed to the Workers, 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; see Section 5.6. Note that 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.

Copyright © 2013 R20/Consultancy, all rights reserved.

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, database technology, and data virtualization. He works for R20/Consultancy (www.r20.nl), a consultancy company he founded in 1987.

Rick is chairman of the annual European Data Warehouse and Business Intelligence Conference (organized in London). He writes for the eminent B-eye-Network.com[13] and other websites. He introduced the business intelligence architecture called the *Data Delivery Platform* in 2009 in a number of articles[14] all published at BeyeNetwork.com.

He has written several books on SQL:

- 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

Published in 1987, his popular *Introduction to SQL*[15] was the first English book on the market devoted entirely to SQL. After more than twenty years, this book is still being sold, and has been translated in several languages, including Chinese, German, and Italian. His latest book[16] *Data Virtualization for Business Intelligence Systems* was published in 2012.

For more information please visit www.r20.nl, or email to rick@r20.nl. You can also get in touch with him via LinkedIn and via Twitter @Rick_vanderlans.

About Teradata and the Teradata Aster Discovery Platform

Teradata Corporation is the world's leading analytic data solutions company, focused on integrated data warehousing, big data analytics, and business applications. Teradata's innovative products and services deliver data integration and business insight to empower organizations to make the best decisions possible and achieve competitive advantage. Visit teradata.com for details.

The Teradata Aster Discovery Platform enables organizations to accelerate analytic innovation and competitive advantage by unlocking new value in big data. It is the market leading discovery platform for big data that provides a complete solution for visual, interactive, fast big

[13] See http://www.b-eye-network.com/channels/5087/articles/
[14] See http://www.b-eye-network.com/channels/5087/view/12495
[15] R.F. van der Lans, *Introduction to SQL; Mastering the Relational Database Language*, fourth edition, Addison-Wesley, 2007.
[16] R.F. van der Lans, *Data Virtualization for Business Intelligence Systems*, Morgan Kaufmann Publishers, 2012.

Copyright © 2013 R20/Consultancy, all rights reserved.

analytic applications that require minimal time and effort. The Teradata Aster Discovery Platform comprises of Teradata Aster Database and Teradata Aster Discovery Portfolio.

Copyright © 2013 R20/Consultancy, all rights reserved.

Appendix A The Built-in Functions of Teradata Aster Database

This appendix contains a list of functions supported by Teradata Aster Database. 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 **Cfilterviz:** advanced visualizations to identify the affinity between two products, item, entities using sigma graphs **Npathviz:** advanced visualizations to interactively see specific paths or groups of paths that a customer has taken to reach an end goal (e.g., service cancellation) **Attribution:** attributes a value to various touch points in a customer's journey (online, offline, or both) towards completing a goal (e.g., product purchase, conference attendance, campaign response)
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 3 *Examples of built-in functions supported by Teradata Aster Database (Source Teradata) – Continues on the next page.*

Copyright © 2013 R20/Consultancy, all rights reserved.

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	**Sessionization:** identifies sessions from time series data in a single pass over the data **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 **XML Parser:** extracts, for example, element name, attribute value, and text from XML documents. XML data are semi-structured and parsers create structured content out of it **SAX:** provides a symbolic representation of time series data **IPGEO Mapping:** takes an IP address and provides the geographical region where the IP is located

Table 4 *Examples of built-in functions supported by Teradata Aster Database (Source Teradata) - Continuation of the previous page.*

Copyright © 2013 R20/Consultancy, all rights reserved.