# Hadoop and Hive as scalable alternatives to DBMS business intelligence solutions for Big Data.

Marissa R. Hollingsworth

November 30, 2011

# 1 Introduction

The persistent evolution of digital technologies and ever-increasing generation, acquisition and storage of digital information allows enterprise and research organizations to uncover novel insights into information that were previously impossible to decode. The amount of digital information generated (and replicated) by individuals on the internet, governments, organizations, research laboratories, and both enterprise and small businesses continues to increase at an annual compound rate of about 60% [**?**][**?**]. In 2009, the amount of digital information worldwide reached a record of 800 *billion* gigabytes and according to an IDC study, the amount of digital information generated in 2010 was an estimated 1,203 exabytes [**?**][**?**], which means approximately 171 gigabytes of data per person per year! *This* is what technologists, analysts, and executives refer to as *Big Data*.

What is the significance of this Big Data explosion? Why do we want to store *all* of this digital information? Take, for example, the major algorithm used in the Google search engine: Google PageRank. The major success of PageRank didn't come from the algorithm itself, but rather from adding additional information (hyperlinks and anchortext) to their dataset [**?**]. This is just one of many cases where "more data...beats better algorithms" [**?**]. The success of many large corporations such as Google, Yahoo, Microsoft, Wal-mart, Disney, and the New York Times, and major research organizations such as the U.S. National Aeronautics and Space Administration (NASA), the Sloan Digital Sky Survey, and the U.S. National Oceanic and Atmospheric Administration (NOAA) [**?**][**?**], depends on successfully analyzing Big Data.

Until recently the high-cost and challenges presented by the storage, accessibility, and scalability of Big Data analysis has discouraged small businesses from taking advantage of the business intelligence insights it can provide. When data sets fit on a single machine, the traditional methods of storing and analyzing data using DBMS are widely available at a relatively low cost to setup and manage. However, as data sets grow the cost of the database approach increases non-linearly. Hadoop MapReduce is an alternative approach to managing data that is linear in cost and can easily handle larger datasets.

Effective business intelligence solutions greatly benefit from correlations between large amounts of archived data from multiple databases [**?**][**?**]. However, the "one size fits all"

approach of existing DBMSs does not work well with data analysis, largely due to the underlying write-optimized "row-store" architecture [**?**]. While write-optimization allows for efficient data import and updates, the design limits the achievable performance of historical data analysis that requires optimized read access for large amounts of data.

Another drawback of the traditional database approach stems from the lack of scalability and fault-tolerance as the number of stored records expands. For scalability, we can move data to a parallel DBMS system, but a successful processing engine for Big Data should be easily scalable, fault-tolerant and must have some elements of a DBMS, an application server (to efficiently process data in parallel) and a message passing system (for distributed storage access and data processing) [**?**][**?**]. While the parallel DBMS provides only some of these elements, the Apache Hadoop project provides all of the above.

In their research, Pavlo et al. found relative performance advantages of some parallel database systems over Hadoop [**?**]. However, they also found that the "up-front cost advantage", ease of set-up and use, and superior minimization of lost work due to hardware failure shines in comparison to the overhead cost of the databases systems. Since our case study calls for a relatively simple and easily scalable solution, we choose to use Hadoop because of its low cost, scalability, fault-tolerance and quick retrieval features for data management.

## 1.1 Hadoop

Hadoop provides several open-source projects for reliable, scalable, and distributed computing [**?**]. Our project will use the Hadoop Distributed Filesystem (HDFS) [**?**], Hadoop MapReduce [**?**], and Hive [**?**].

### 1.1.1 MapReduce for Data Processing

Hadoop provides a programming model known as MapReduce that abstracts the problem into computations over key and value sets [**?**]. The model works well with Big Data because it is inherently parallel and can easily handle data sets spanning across multiple machines.

Each MapReduce program runs in two main phases: the map phase followed by the reduce phase. The programmer simply defines the functions for each phase and Hadoop handles the data aggregation, sorting, and message passing between nodes.

**Map Phase.** The input to the map phase is the raw data. A map function should prepare the data for input to the reducer by mapping the key to the the value for each "line" of input. The key-value pairs output by the map function are sorted and grouped by key before being sent to the reduce phase.

**Reduce Phase.** The input to the reduce phase is the output from the map phase, where the value is an iterable list of the values with matching keys. The reduce function should iterate through the list and perform some operation on the data before outputting the final result.

There can be multiple map and reduce phases in a single data analysis program with possible dependencies between them.

### 1.1.2 Hive for Data Warehousing

Hive is a data warehouse system for Hadoop that facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in Hadoop compatible file systems [**?**]. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL. At the same time this language also allows traditional map/reduce programmers to plug in their custom mappers and reducers when it is inconvenient or inefficient to express this logic in HiveQL [**?**].

## 1.2 A Big Data Problem: Payment History Analysis

Many companies with payment plan options can reduce the amount of resources required to collect payments by predicting whether a customer will make their payments on-time, late, or never.

For this project, we will consider a small, but expanding, company (which we will also refer to as our *consultant*) that offers payment plan options to its customers so that they can pay for the provided services over time. While payment plans benefit both the company and customer by allowing the company to provide a service and the customer

to use a service without having the funds to pay the total charges up front, the company takes the risk of liability for late or lost funds due to a customer's inability to pay on-time. However the liabilities of these risks can be reduced by using past payment patterns to predict whether to expect on-time payment, extend the payment period for the customer, or write off charges. These predictions can reduce administrative costs such as outsourcing the collection of charges for late-paying customers who will eventually pay and keeping records of charges that will never be collected.

Our consultant has successfully implemented this payment analysis scenario in MySQL (see Appendix A), but the company expects to see a significant increase in the number of account records as they expand their client base over the next year. This is a concern because, as mentioned in previous sections, the DBMS model does not scale as the number of accounts increase and will no longer work when the data does not fit on a single system. Due to the limited resources of the company, the solution must be low-cost and easy to implement and maintain.

# 2 Project Statement

The proposed project will study the feasibility of a scalable solution to the payment analysis case study while providing a basic model for developing Big Data business intelligence applications using Hadoop. The goals of the project are as follows.

- Design and implement a scalable business intelligence solution for extracting patterns in customer history data.
- Generate large sample data sets (hundreds of gigabytes) using Hadoop to compare the scalability of the MapReduce solution to the existing MySQL solution.
- Implement and compare several Hadoop I/O techniques to achieve an optimized MapReduce solution.
- Implement and compare a solution in Hive.

The model will include a detailed outline of the design and implementation stages followed by a comparison and analysis of the performance, ease of implementation, and manageability of the MapReduce, Hive and the original DBMS implementations.

# 3 Methods

The development cycle of the proposed project will generally adhere to the *Agile* software development model where the design, implementation, testing, and analysis phases will occur simultaneously and remain flexible to adjustments as the project progresses. The following sections outline the primary goals of each phase.

## 3.1 Requirements Phase

The majority of the requirements phase will occur before the other phases begin. The objectives of this phase are as follows.

1. Meet with consultant to discuss overall project requirements and specifications.
2. Clearly outline the details and constraints of project input and expected output.
3. Work with consultant to verify details and constraints before beginning the design phase.
4. Consistently verify that the requirements are being met and remain applicable during all phases of the project.

## 3.2 Design Phase

After the project requirements have been finalized, we will enter the design phase during which class diagrams, algorithms, and job flows will be defined. While the details of these designs will be comprehensive, it is important to allow modifications during the implementation, test and analysis phases as required. We will blueprint the following components throughout the design phase.

- **Custom writable classes**. In order to pass multi-attribute values (*n-tuples*) between map and reduce functions, Hadoop requires that the representing objects implement the `WritableComparable` interface. We will need to implement `Writablecomparable` for the following classes.
    1. `Customer` - object representing customer tuple.
    2. `Account` - object representing an account tuple.
    3. `Transaction` - object representing a transaction tuple.

4. `StrategyHistory` - object representing a strategy history tuple.

- **MapReduce job flow**. The complexity of the payment analysis problem requires several stages of data aggregation to achieve the final results. We will need to design the details and algorithms to achieve the output for the following jobs.

    1. `GenerateAccountStatistics` - aggregate transaction totals per account.

        **input**: all of the `Transaction` and `StrategyHistory` records.

        **output**: an object storing the aggregate adjustment, charge, and payment sums for each account.

    2. `AccountsPerCustomer` - group account details by customer.

        **input**: the account details output by the `GenerateAccountStatistics` job and the `Account` records.

        **output**: a `Customer` object with the customer attributes and account details per customer.

    3. `CustomersPerSsn` - group customer details by social security number.

        **input**: output from the `AccountsPerCustomer` job and the *Customer* records.

        **output**: a `Customer` object containing the customer attributes and a list of all the accounts owned by the customer per SSN.

    4. `AccountsPerSsn` - group customers by `SSN` so we can calculate the previous account values for each account. (A previous account is defined as an account with OpenDate prior to this Account's OpenDate which has a Customer with the same SSN).

        **input**: output from the `CustomersPerSsn` job.

        **output**: a new `Account` object with the account attributes and previous account attributes set.

- **Hive data set structure**. The schema used to store the data in HDFS can have profound effects on the efficiency of Hive queries. We will use the implications of Gupta et al.'s research on efficiently querying archived data using Hadoop to design our storage schema [**?**].

## 3.3 Implementation Phase

We will use the following details as we implement the established program design.

**Hadoop and Java APIs**. Our implementation of the payment analysis application will use the "old" Hadoop API (releases prior to 0.20 series) and the Java 1.6 API. While the new Hadoop API has been released in the 0.20 series, we choose not to use it because it is known to be incomplete and unstable.

**Tools**. The majority of the development will be in Eclipse using the MapReduce plug-in for accessing the HDFS. We will use MySQL Workbench to load and execute the MySQL version of the payment analysis solution for benchmarking.

**Implementation steps**. The implementation of the proposed project will procede in the following order.

1. MapReduce solution to the payment analysis case study.
    - Develop object classes to represent tuples.
    - Develop map and reduce functions for each job.
    - Chain jobs to achieve final results.
2. Sample data generation for benchmarking the results in the test phase.
3. Hive solution to the payment analysis case study.
    - Configure the HDFS for use with Hive.
    - Translate MySQL solution to the HiveQL dialect.

## 3.4 Test Phase

Because we will follow an agile program development model, testing will occur throughout the entire development process, using the techniques listed below.

- **Unit testing**. To ensure the accuracy of our results as application development progresses, we will maintain a test suite by adding new test cases for each job and object class we develop. Since we have the desired outcome for the payment analysis case study, we will compare the final results of each solution to the expected results.

- **HDFS cluster setup**. Most testing during development will occur on a pseudo-distributed HDFS cluster. However, when benchmarking the results on large data sets, we will manage and run our results on a fully-distributed HDFS cluster. We will run the benchmark testing on HDFS clusters with varied numbers of data nodes.

- **Data set generation**. Since we are testing the scalability of our solution for Big Data, we need to generate large data sets; ranging between 10GB and 500GB.

# 4 Project Schedule

| | |
|---|---|
| December 2010 | - Meet with consultant to define problem. |
| January 2011 | - Obtain specification documents and start application design phase. |
| February 2011 | - Solidify application requirements and design.<br>- Begin implementation and test phases of MapReduce solution. |
| March 2011 | - Finalize MapReduce solution.<br>- Begin implementation phase of sample data generation. |
| April 2011 | - Use sample data to compare MapReduce implementation to MySQL solution.<br>- Begin implementation and test phases of Hive solution.<br>- Write report sections for MapReduce solution. |
| May 2011 | - Finalize Hive solution.<br>- Use sample data to compare Hive implementation to MapReduce and MySQL implementations.<br>- Write report sections for Hive solution.<br>- Finalize report. |

# Appendix A

Payment Analysis specifications

**Input**. Text files containing data extracted from the following tables in the existing MySQL database.

- **Customer** - The *CustomerNumber* is unique for each *Customer*. The *SSN* may be the same for multiple customers representing the same person. The *ZipCode3* value represents the first three digits of the customer's zip code.

| CUSTOMER | | | | |
|---|---|---|---|---|
| CustomerNumber | FirstName | LastName | SSN | ZipCode3 |

- **Account** - The *AccountNumber* is unique for each *Account*. The *CustomerNumber* may be the same for multiple accounts owned by the same customer. The *OpenDate* of the account is the date the account was opened by the customer.

| ACCOUNT | | |
|---|---|---|
| AccountNumber | OpenDate | CustomerNumber |

- **Transaction** - The *AccountNumber* is a foreign key referencing the *Account* to which the transaction belongs. The type of transaction is represented by the *TransactionType* and has a value of *charge*, *adjustment*, or *payment*. The *TransactionDate*, *TransactionAmount* hold the date and amount of the transaction.

| TRANSACTION | | | |
|---|---|---|---|
| AccountNumber | TransactionDate | TransactionAmount | TransactionType |

- **Strategy History** - The *AccountNumber* is a foreign key referencing the *Account* to which the strategy history entry belongs. The name of the strategy is represented by *StrategyName* and has a value of *Good Standing* or *Bad Debt*. The *StrategyStartDate* is the date the the account status switched to *StrategyName*.

| STRATEGY HISTORY | | |
|---|---|---|
| AccountNumber | StrategyName | StrategyStartDate |

**Output**. The output of the program will be a table with the following entries.

- *AccountNumber* - unique Account identifier.
- *OpenDate* - date the Account was opened by Customer.
- *ZipCode3* - first three digits of Customer's zip code.
- *TotalCharges* - sum of all Charge transactions on Account.
- *TotalAdjustments* - sum of all Adjustment transactions on Account.
- *AdjustedTotalCharges* - sum of TotalCharges and TotalAdjustments.
- *TotalGoodStandingPayments30Day* - negative of sum of all Payment transactions on account between 1 and 30 days after "Good Standing" Strategy start date but before "Bad Debt" Strategy start date.
- *TotalGoodStandingPayments60Day* - same as above, but between 1 and 60 days.
- *TotalGoodStandingPayments90Day* - same as above, but between 1 and 90 days.
- *BadDebtTransferBalance* - sum of all transactions on account up through "Bad Debt" Strategy start date.
- *TotalBadDebtPayments30Day* - negative sum of all Payment transactions on account between 1 and 30 days after "Bad Debt" Strategy start date.
- *TotalBadDebtPayments60Day* - same as above, but between 1 and 60 days.
- *TotalBadDebtPayments90Day* - same as above, but between 1 and 90 days.
- *PreviousAccountCount* - number of Accounts with Open Date prior to this Account's Open Date which have a Customer with the same SSN.
- *PreviousAccountGoodStandingCharges* - sum of all Charge transactions occurring prior to this Account's Open Date on Accounts which have a Customer with the same SSN (not same account) which occurred while other Accounts were at or after "Good Standing" Strategy start date but before other Accounts were at "Bad Debt" Strategy start date.

- *PreviousAccountGoodStandingAdjustments* - sum of all Adjustments that occurred during Good Standing strategy for past accounts.
- *PreviousAccountGoodStandingPayments* - sum of all Payments that occurred during Good Standing strategy for past accounts.
- *PreviousAccountBadDebtPayments* - sum of all Payments that occurred during Bad Debt strategy for past accounts.