# Two Parallel Approaches to Network Data Analysis

Arian Bär
FTW
Vienna, Austria
baer@ftw.at

Antonio Barbuzzi    Pietro Michiardi
EURECOM
Sophia-Antipolis, France
{barbuzzi, michiard}@eurecom.fr

Fabio Ricciato
FTW
Vienna, Austria
ricciato@ftw.at

## ABSTRACT

In this work we compare two alternative approaches to large-scale analytic applications. We focus on network data analysis and define four sample Jobs that operate over a publicly available dataset of a trans-Pacific Internet link.

First, we present an approach based on a shared-nothing parallel database and discuss the key ingredients of its design. Then we present an approach based on MapReduce, with focus on the design of data analysis Jobs and their optimization.

Besides a mere performance comparison, the lessons we learned from several experiments with such systems highlight the challenges in performing the same computations over the same datasets, with two orthogonal approaches.

## 1. INTRODUCTION

The endeavor of this work is to compare alternative approaches to data-intensive analytics applications, which involve batch processing of large amounts of data with a parallel system. Although it would be tempting to proceed with a general comparison, as done in the much discussed articles [7, 9], in this work we start from a concrete problem and focus on a specific application context, namely network data analysis. In such applications, the goal is to analyze a vast amount of data collected from one or more vantage points in the network and stored in a centralized location. Network data is historical in nature, which imply a simple *write once, read many* workload: traces are timestamped and, once a dataset is produced, no updates are required. Applications of network data analysis include anomaly detection, traffic classification, botnet detection, quality of service monitoring and many more. In a typical setting, network data are periodically accumulated in a storage system for future processing and analysis.

Therefore, data import operations are required to complete in a time that is strictly less than the accumulation period. The subsequent processing phase, especially when the volume of data to analyze is large, may not be feasible with traditional network analysis tools [6]: in this work we advocate for two alternative parallel approaches to data analysis.

The first method, which we call TicketDB, is based on a shared-nothing parallel database system design. Despite the vast amount of literature on the subject (see e.g. [5]), shared-nothing architectures have been recently popularized by Teradata, as discussed in [9], and are used today by many Internet services, like e.g. eBay. Our design features specific data management and partitioning mechanisms for network data analysis. The second method which we consider uses the MapReduce framework [4], and specifically its open-source implementation that is part of the Hadoop project [1]. Both approaches are detailed in Section 2.

In this work, we proceed with the comparison of these two parallel systems by first defining a set of data analysis Jobs, which are detailed in Section 3. Our Jobs are representatives of typical (*e.g.,* from a Network Operator perspective) network analysis tasks related to users' and protocol behavior. They are run on a large, publicly available network trace from a trans-Pacific Internet link. One of the main contributions of this article is to specify how such Jobs can be described and implemented in each of the two considered systems. More to the point, the aim of our experiments goes beyond a mere head-to-head performance comparison of two parallel systems.

First, we show that with TicketDB, most of the effort is devoted to the system design, as there is no publicly available community-supported open-source project to a shared-nothing parallel database system. We designed such system from scratch, including the auxiliary software components to load network data into different instances of the underlying DB engine, and for data partitioning tailored to network data analysis. With this approach, Jobs are expressed with SQL, which greatly simplifies the task of defining the operations on network data.

Then, we focus on MapReduce and describe the large number of "knobs" that can be tuned in order to reach reasonable system performance. As opposed to TicketDB, with MapReduce the system design requires practically no effort — besides deployment issues. Instead, we show that the design and implementation of analysis Jobs is more involved and requires great zeal. In this work, we present a novel technique for time-stamped data whose aim is to produce a compact execution plan for jobs that otherwise would require a complex and lengthy sequence of intermediate jobs.

For sake of completeness, in Section 4 we report on the performance of the two parallel alternatives and conclude in

Section 5 with a series of lessons learned from our work.

## 2. SYSTEM ALTERNATIVES

In the following, we proceed with a description of the design and implementation of the two systems. The technical details of each approach are tailored to the nature of the data at hand and related operations.

First, we focus on TicketDB and sketch the salient features of the architecture and main components. Then, we outline the Hadoop system: we gloss over several details of MapReduce and focus on job optimization and important tuning parameters.

**TicketDB:** The requirements that motivate the design of a parallel database from scratch stem from the particular application we study in this work. Due to the ever-increasing speed of computer networks, the import operation consisting in copying data to the centralized database must be very efficient. Moreover, network analysis may involve querying and processing a vast amount of data, hence query performance is also critical.

We note that network data is historical in nature: traces are timestamped and, once a dataset is imported in the database, no updates are required. Thus, we consider network data analysis to belong to Online Analytical Processing (OLAP) applications, and adopt traditional OLAP techniques to organize the data. We use the star schema[2]: we split "dimensions" from "fact" data, and use a two-way table partitioning, which allows to run queries in parallel on many partitions, with a final aggregation. Dimension and fact data are stored into separate tables which are referenced via a unique identifier. For every entry in the dimension table there are one or more entries in the fact table.

The architecture of TicketDB is sketched in Fig. 1. In our current implementation, TicketDB embraces the "scale-up" paradigm, as it is deployed on a single, high-end machine, provisioned with a large number of computing cores, memory and storage arrays.

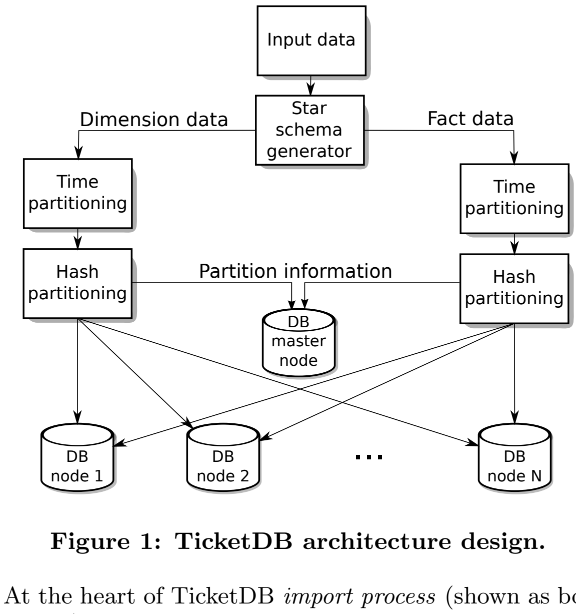

**Figure 1: TicketDB architecture design.**

At the heart of TicketDB *import process* (shown as boxes in Fig. 1) lay two components that perform time and hash data partitioning. First, data is *range partitioned* based on the time information included in the data — on a per-hour basis in our case. Each time-partition is then further *hash partitioned* based on the unique identifier from the dimension table — in the currently deployed system we use two hash partitions. Every hash partition is stored in a separate database instance, with its own RAID array for data storage. Data partitioning information, which tracks the table to database instance mapping, is stored in a specific table on the master node. We note that partitioning has the following benefits: *(i)* indices can be created on individual partitions and are therefore small; *(ii)* after the full import of a partition, indices can be created in parallel to the import of the next partition; *(iii)* queries can run in parallel on multiple partitions at the same time; and finally *(iv)* full partitions can be dropped when data has to be deleted.

In the current design of TicketDB, each database instance is a PostgreSQL engine residing on the same physical machine. Although data redundancy is achieved through the RAID-6 disk array attached to the machine, which can overcome two disk failures, our current implementation does not support replication nor failure detection. If the physical machine fails, the whole system becomes unavailable. Future extensions of the TicketDB design will address replication and failure detection features.

**Hadoop:** MapReduce, popularized by Google with their work in [4], consists in both a programming model and an execution framework that is deployed on a cluster of machines connected by a local area network. In MapReduce, a Job consists of three phases and accepts as input a dataset, appropriately partitioned and stored in a distributed file system. In the first phase, called MAP, a user-defined function is applied, in parallel, to input partitions to produce intermediate data, which are stored on the *local* file system of each machine of the cluster. Intermediate data is sorted and partitioned when written to disk. Next, during the SHUFFLE phase, intermediate data is "routed" to the machines responsible for executing the last phase, called REDUCE. In this phase, intermediate data from multiple mappers is sorted and aggregated to produce output data which is written back to the distributed file system. Complex Jobs may require several iterations or combinations of MAP, SHUFFLE, REDUCE phases.

Now, we focus on the key ingredients that define the performance of a Job in terms of execution time. Simply stated, disk and network I/O are the main culprits of poor Job performance. The task of a Job designer is thus to optimize the amount of memory allocated to mappers and reducers, so as to minimize disk access. Moreover, a Job may include an optional COMBINER phase in which intermediate data is pre-aggregated before it is sent to reducers, to minimize network utilization. Job optimization is a manual process that requires knowledge of the size of intermediate data sent to each reducer, and the characteristics of the cluster, including number of nodes, number of processors and cores and available memory.

## 3. NETWORK DATA PROCESSING JOBS

Here we briefly describe the analyzed datasets, and provide a high-level description of the four sample Jobs. In Sec. 3.1 and Sec. 3.2 we describe how Jobs are defined in our implementation of TicketDB and MapReduce respectively.

**Network Data:** The input to our Jobs is a textual CSV file of network traces from the publicly available MAWI (Measurement and Analysis on the WIDE Internet) archive, captured from a trans-Pacific link between the United Stated and Japan [3]. In particular, we use traces from samplepoint F collected from 2009/03/30 to 2009/04/02. Packet payload is omitted and IP adresses are anonymized in the original trace. From it we extracted a CSV file consisting of one record per packet. Each record has a size of approximately 64 B and includes information like timestamp, source and destination IP addresses and ports, flags, used protocol and packet length. The original trace consists of 432 GB raw data, resulting in a CSV file of roughly 100 GB.

**Sample Jobs:** Hereafter we describe four sample Jobs representative of typical analysis tasks run by network operators. Each Job compute statistics hourly, daily and for the whole trace duration.

- **User Ranking (J1).** For every IP address, compute the number of uploaded, downloaded and total exchanged (upload + download) bytes.
- **Frequent IP/Port by Heavy Users (J2).** Heavy users are defined as the top-10 IP addresses from Job J1 in terms of total exchanged bytes. This Jobs computes the top IP/Port pairs for all heavy users.
- **Service Ranking (J3).** For experimental purpose, the set of IP addresses is split arbitrarily into two subsets, $\mathcal{A}$ (servers) and $\mathcal{B}$ (clients). For every IP address in $\mathcal{A}$, this Jobs computes the total number of connections established by IP addresses in $\mathcal{B}$.
- **Unanswered TCP Flows (J4).** For every IP address, this Job computes the number of unanswered TCP flows. A TCP flow is considered *answered* if the initial SYN packet is acknowledged by a SYN-ACK within 3 seconds.

## 3.1 Job Implementation in TicketDB

In the following we describe how data is imported and queried in TicketDB. At the end of the section we give a detailed description of how Job J1 can be expressed using the stored procedures of TicketDB (due to space limitation we do not provide details for the remaining Jobs).

**Import:** The data is imported into TicketDB using a loader program written in C. The loader first splits the data into *dimension data* (*i.e.* the IPs, ports, protocol) and *fact data* (*i.e.* timestamp, flags, packet/byte counters). Dimension and fact data are imported into separate tables which are referenced via a unique identifier. Hash partitioning contributes to substantial improvement during the data import phase because multiple partitions can be written in parallel. In addition, queries can run on all hash partitions in parallel, mapping the data to a subset, which then can be merged or stored in new tables for further processing. Note that data is not loaded into the database using inserts, but first written into a temporary file residing in the shared memory, which is then imported into the database every 100k lines by the `copy` command. In this way, import speed is increased substantially.

The physical storage size of the range partitions has a big impact on the query performance. If the size of a range partition exceeds the memory available for a database connection, sort operations, normally performed in memory, require to save intermediate data on disk and are therefore less efficient. The amount of data for a given time interval changes over time due to the time-of-day variations of network data. Therefore, the time interval for a range partition has to be choosen that small, that the physical storage size does not exceed the available memory even in the peak hours.

**Queries:** The queries to TicketDB can be run in parallel on multiple range and hash partitions at the same time. How this is achieved can be explained with an illustrative example. Assume we want to compute the total number of bytes exchanged on this link on a day, as described in Job J1. In our database setup, for every hour we have two hash partitions stored in two separate nodes. This leads to a total of 48 partitions to be queried for one day.

First, for each hourly partition we create a new *outer* connection to the database. Then, inside each of those connections we create a new *inner* connection for each hash partition. Inside the inner connection, we compute the sum of all packet length fields and return the result to the outer connections. Each outer connection returns the sum of the results from both inner connections. To obtain the final result, the results from all outer connections are summed up. Obviously, the creation of inner and outer connections is fully automatic: we implemented two stored procedures to handle the connection creation that greatly simplifies query writing. One procedure, called `qf_dist_sql_outer`, handles the creation of the outer connections — the number of concurrent connections is configurable. The other procedure, called `qf_dist_sql_inner`, handles the creation of the inner connections. Thus, we only have to define the query to execute inside the inner connections and how the results of the inner and outer connections should be merged. All the merging is done only by the master node. For the performance evaluation presented later we used the stored procedures for all queries running 24 parallel outer connections.

The query in Figure 2 shows how the hourly results for J1 are calculated in TicketDB using the two stored procedures introduced above. The most inner query collects the uploaded and downloaded bytes per IP address. Therefore, the table containing the packet length (`fact_ip`) has first to be joined with the dimension table (`dim_ip`) containing the IP addresses. The next step is to transform the triple `<source_ip, dest_ip, bytes>` into uploaded and downloaded bytes for each distinct IP address: this is done by the inner union. Then the results are grouped over timestamp (`ts`) and IP address, and the sum of the bytes is calculated. This query is evaluated per hash partition on each database node in parallel. After it has finished, the function `qf_dist_sql_inner` returns the results from each inner connection by using the `union all` operator. This means that we have to group the results again by timestamp and IP address, and sum the bytes once more to create the hourly tables. The function `qf_dist_sql_outer` takes care of running the multiple outer connections. If more partitions than the amount of available parallel connections have to be queried, the query is executed in multiple rounds. In each round all available connections are used and the function waits until all connections complete their query[1]. With respect to Job J1, tables are created for every hourly partition, and no re-

[1]This process can be optimized by reusing the connection as soon as the query is finished. The current implementation does not include this feature, that is left to future development.

```
select * from qf_dist_sql_outer($outer$
 insert into heavy_users_TIMESTAMP (
  select ts, ip, sum(up) up, sum(down) down,
    sum(total) total
   from qf_dist_sql_inner($inner$
    with t as
     (select ts, source_ip, dest_ip, up bytes from
      (select timestamp - (timestamp % 3600) ts,
       id, sum(bytes) up
       from fact_ip group by ts, id) as f
     join dim_ip d on (f.id=d.id))
    select ts, ip, sum(up) up, sum(down) down,
        sum(tot) tot
      from (
       select ts, source_ip ip, bytes up, 0 down,
         bytes tot from t
       union all
       select ts, dest_ip ip, 0 up, bytes down,
         bytes tot from t
      ) foo
    group by ts, ip order by ts, tot desc
   $inner$, TIMESTAMP)
  group by ts, ip)
 )
$outer$, start_timestamp, end_timestamp);
```

**Figure 2: Query for Job J1.** This example illustrates how queries are expressed in TicketDB (due to space limitations we give full details only for Job J1).

sults are returned from the `qf_dist_sql_outer` function. If instead results are returned from the outer connections, a final concatenation and grouping is required. The hourly tables produced in this way are then used in the next step to compute the daily and whole-trace result.

## 3.2 Job Implementation in MapReduce

MapReduce Jobs require to define a Map and a Reduce function, the "routing" of data from mappers to reducers (called Partitioning and Grouping), the order of data received by the reducers and possibly, to further optimize the job, a Combiner. We remark that there are several ways to implement our sample Jobs. Although we have tried many alternatives, in the following we only present best-performing implementations.

Before proceeding any further, we discuss the key ideas underlying the conceptual design of our jobs. As an illustrative example, let's consider the fact that our jobs compute statistics at different time granularities, namely hours, days and weeks. A naive approach is to launch a Job for each time period and, if possible, use the output of Jobs executed on fine grained periods as input for Jobs running on coarse grained periods. However, this approach requires various cycles of data materialization and subsequent dematerialization (not to mention framework overhead), that can be avoided by a careful design of the Reduce, Grouping, Sorting and Partitioning phases.

In this work we use a "design pattern"[2] that we labeled *in-reducer grouping*, whose aim is to produce a compact execution plan instead of using a sequence of jobs. To understand our technique, it is convenient to recall the purpose of the standard Grouping phase in Hadoop. Grouping is used to prepare the input data to the Reduce phase by grouping all intermediate key/value pairs in output from the Map phase that refer to the *same* key. As such, the Reduce phase receives a series of $(key, list < values >)$ for further processing. By default, the Grouping phase in Hadoop is *implicit* and exectued at the framework level.

The gist of our approach is to make the Grouping phase *explicit*, by moving its definition and execution at the user-code level. Fig. 3 gives an high-level overview of the *in-reducer grouping* technique. In the Figure, we show the behavior of a *single* machine executing the Reduce phase. The input is in the usual form of sorted key/value pairs which are processed in sequence (from left to right) by subsequent calls to the reducer function. Our design pattern is used to organize the input to the Reduce phase such that it can be dispatched to different reduce functions: in the Figure, we show two different grouping strategies that, based on the same input, "feed" two different reduce functions $(R_1, R_2)$. It is fundamental to note that grouping keys must follow a hierarchical organization: following up with the motivating example outlined above, one grouping key would be based on hourly data (and sent to $R_1$) and another on days (and sent to $R_2$). Note that in some specific cases, the output of one reduce function in the same Reduce phase can be routed to another reduce function, which may lead to computational cost savings: this is illustrated in the Figure by the dotted lines between $R_1$ and $R_2$. There is a further subtle aspect in our technique, that we omit from Fig. 3 for the sake of clarity. For elements to be grouped by key, a preliminary sorting phase is required. With reference to our example, we define a custom Sorting phase such that keys are sorted according to the hourly data, which is the most fine-grained period we have in our Job. Finally, we also specify a custom Partitioning phase, that sends all data that belongs to the the coarse grained group to the same reducer.

We acknowledge that fact that there are various high-level data query languages for Hadoop (such as Pig, Hive, and JAQL) that are more concise if compared to the plain implementation of a MapReduce job. However, we remark that there is a trade-off between performance and verbosity: in practice, designing jobs in native Java code is considered to produce more efficient executions, as noted in [8]. Moreover, we note that our best-performing jobs cannot be expressed in the declarative style of higher level languages, since such languages cannot be used to modify the inner functioning of the framework itself.

Now, we can move forward and describe the design of each Job defined in Sec. 3.

**Job J1:** For this Job, we need to compute the total bytes sent and received for each IP address per hour/day/whole trace. Therefore, we aggregate data by time-range and IP. In the Map phase, for each record, we emit two records with a composed key `<timestamp, IP>`. The value accounts for the packet size and the direction (sender or receiver). The Reduce phase receives all the data for each IP ordered by timestamp, uses a counter for each time-range, and immediately emits the output per time-range.

This Job requires a Partitioning function based on the IP that assigns all the records with the same IP to the same reducer, and a Sorting function that sorts the data based

[2]The "design pattern" we discuss is applied to timestamped datasets. We are currently working on a generalization of this technique for other kind of data.

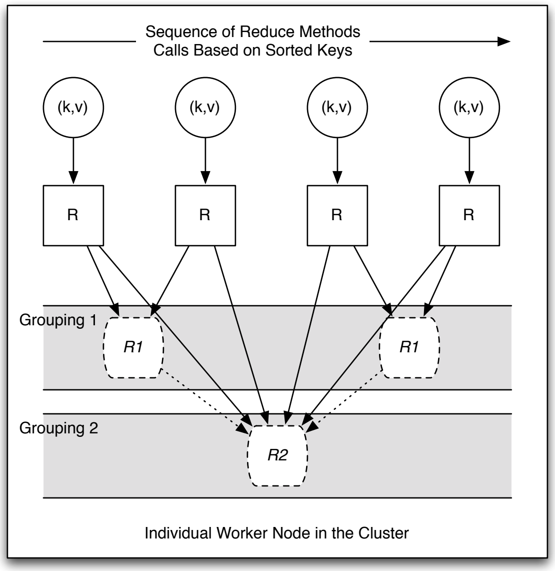
Sequence of Reduce Methods Calls Based on Sorted Keys
(k,v)
(k,v)
(k,v)
(k,v)
R
R
R
R
Grouping 1
R1
R1
Grouping 2
R2
Individual Worker Node in the Cluster

**Figure 3: An illustration of the *in-reducer grouping* technique.**

on the IP as the primary key, and the timestamp as the secondary key.[3]. As opposed to a naive approach, this Job reduces scheduling overhead of the framework, disk and network I/O, but is more complex to write.

**Job J2:** This Job requires two phases: *(J2.1)* find heavy users, and *(J2.2)* find top ports of heavy users.

The input data of Job J2.1 is the output of Job J1. We use an hash table of fixed-size priority queues in the Map phase, that stores only the top 10 users for each input data block. In the Reduce phase, we compute the heavy users over all intermediate data from mappers.

In Job J2.2 we need to read the input CSV file, and emit a record only for heavy users. Hence, we use a distributed cache to save the list of heavy users, so that it is locally available to each Map function. During the setup phase of each mapper, we read the contents of the distributed cache and load them in a hash map. It is important to note that the top-10 ports per day cannot be calculated from the top ports per hour: the “top” operation is not distributive. Job J2.2 is a single Job, where for each record containing an IP from a heavy user, the Map emits three records, using the composite key `<time-range, IP>` and the port and the size as value. The Reduce receives all the records belonging to the same composite key: a fixed size priority queue is used to emit top ports per distinct key. We tested four alternative approaches to implement Job J2.2. In general, we noticed problems related to memory requirements in the Reduce phase and volume of intermediate data transmitted in the Shuffle phase. The approach described above is the one that, in our experiments, performed better.

**Job J3:** Given two subsets $\mathcal{A}, \mathcal{B}$, we find the total number of unique IPs in $\mathcal{A}$ contacted by IPs in $\mathcal{B}$, as follows. For each record in the input file whose source IP is in $\mathcal{B}$ and whose destination IP is in $\mathcal{A}$, we emit a record containing the IP address in $\mathcal{A}$, the IP address in $\mathcal{B}$ and the time-range which the record falls into.

In the Map phase, we emit one record with a composite key `<IP` $\in \mathcal{A}$ `, IP` $\in \mathcal{B}$`, timestamp>` for each input record from the CSV file. This choice of composite key simplifies the Reduce function: instead of (naively) using a simple key `<IP` $\in \mathcal{A}$`>`, we exploit the sorting capabilities of the framework to achieve our goal. The Reduce function receives all packets belonging to the same composite key and counts the number of unique IP in $\mathcal{B}$. This approach requires a custom Partitioning function that operates on IP address in $\in \mathcal{A}$ and a custom Sorting function on the whole composite key.

Among all alternative implementations of this Job we experimented with, this approach is the hardest to code, but uses only a single scan of the input data, and produces a small quantity of intermediate data for the Shuffle phase, which also lowers sorting effort. However, it requires more computational resources in the Reduce phase.

**Job J4:** To find the unanswered SYNs for each IP address we need to examine the whole dataset, filtering the records with SYN and SYN-ACK flags. This can be easily done in the Map phase. In order to find if a SYN is unanswered, the Reduce function receives all the SYNs and all the SYN-ACKs having respectively the same source and destination IP address in the same time-range.

The Map phase behaves as follows. For each SYN packet it emits a record with a composite key `<SrcIP, DstIP, SrcPort, DstPort, Timestamp>`, and SYN as value. For each SYN-ACK packet, it emits an “inverted” record with a composite key `<DstIP, SrcIP, DstPort, SrcPort, Timestamp>`, and SYN-ACK as value. The Reduce function receives the SYN and all the possible subsequent SYN-ACK in order, so that it can easily check if the SYN-ACK is present.

Using a custom Partitioning based on the IPs and ports only, all the packets belonging to the same, bidirectional, flow are sent to the same reducer. Using a custom Sorting function based on the whole key, the Reduce function receives all the packets belonging to the same bidirectional flow ordered by timestamp, so that it is easy to check if a SYN is followed by a SYN-ACK.

## 4. EVALUATION

We now present a preliminary evaluation of the two approaches of this work. Our experiments and results should not be interpreted as an head-to-head comparison of parallel databases against MapReduce. Rather, our intent is to understand if, why and to what extent Job completion times differ using the two systems.

As a general remark, one obvious source of discrepancy in the results is due to the asymmetry of the hardware configuration for the two systems. TicketDB runs on a high-end server with two hexa-core Intel XEON 2.93 GHz CPUs and 48 GB of RAM. The operating system is stored on two 500 GB RAID-1 disks. For the database two storage arrays are attached to the server via Fiber Channel links. Each storage array has twelve 750 GB RAID-6 disks. The MapReduce system is based on Hadoop 0.21 running on a cluster of 11 machines deployed on Amazon EC2. Each instance has 7.5 GB of RAM, 2 virtual cores, one 850 GB disk and a 10 Gigabit Ethernet. We note that, considering solely the bare-bone hardware purchase costs estimation — i.e., ignoring energy, cooling and maintenance costs — the two deployments we use in our experiments have roughly comparable costs, of

[3]An additional detail: we use a hour-based Grouping comparator that simplifies the Reduce function.

| Job | MapReduce | TicketDB |
|---|---|---|
| Loading data | 26 min 34 s | 102 min 33 s |
| J1 | 26 min 32 s | 14 min 40 s |
| J2 | 35 min 45 s | 6 min 54 s |
| J3 | 26 min 57 s | 7 min 33 s |
| J4 | 18 min 18 s | 9 min 39 s |

**Table 1: Summary of Job duration.**

about 15 K€. Of course, costs scales differently for the two systems, but a detailed analysis of cost scaling is beyond the scope of this article.

Table 1 summarizes our results, indicating the duration of each Job and the time required to initially load the data. As a baseline benchmark, we evaluate the aggregate I/O to read input data in both systems, which includes hardware limitations and software overheads. TicketDB peaks at 650 MB/s while the figure for Hadoop is roughly 200 MB/s: this partially explains the difference in Job duration between the two systems. Note also that we don't have control over the placement of virtual machines in the Amazon cluster: it is possible for multiple virtual machines to be launched on the same physical machine, which would create additional concurrency issues for the Hadoop system.

One clear advantage of TicketDB with respect to Job duration is due to the fact that a full scan of input data is performed solely during the import phase. Since data is organized into multiple tables, some Jobs can benefit from accessing only a fraction of the tables. This is the case e.g. for J2, which in fact yields the largest difference in completion time between the two systems. Note however that other Jobs (e.g., J1 and J4) may still require access to all tables.

## 5. LESSONS LEARNED

In this work we considered two alternative approaches to large-scale network data analysis. Our goal was to design and compare two parallel systems for processing large amounts of network data, based on a shared-nothing design principle. To this end, we presented TicketDB, a parallel database system, and an alternative system based on MapReduce. Our experience can be summarized as follows.

With TicketDB, we designed the system from scratch: thus, a major development effort has been put in building key components such as data management and partitioning. The SQL statements that implement the sample analysis Jobs considered in this work, albeit not trivial, are more concise than their MapReduce counterparts.

With Hadoop, we put a major effort in the design of MapReduce Jobs, which also involved "bending" the underlying framework for computations to be more efficient and less resource hungry. For complex operations, the same Job could be expressed in different ways: our experience showed that the naive approach of concatenating simple Jobs was far from being efficient. Instead, a common "design pattern", that we called *in-reducer grouping*, of best performing implementations involved using MapReduce as system to partition and "route" data to complex REDUCE functions. Moreover, a substantial amount of work has been devoted to the optimization of Hadoop, by appropriately tuning its several "knobs" to minimize I/O operations, which are largely responsible for slow Jobs.

In conclusion, we believe this work has shed lights on why large-scale analytic applications require a deep knowledge of both system and algorithmic aspects of data processing. Besides the absolute performance of the two alternative (and in some sense orthogonal) approaches, it is important to consider the ramifications of early design choices for batch-oriented processing systems. We showed that although MapReduce has the potential of decoupling system design from the actual data analysis, the reality is that the boundary between these tasks is often blurred, especially when optimizations are required. We also showed that, despite the lack of open-source implementations of parallel databases, the design of the key component of a shared-nothing architecture is not a daunting task: however, the expressiveness of standard query languages such as SQL is not immediate to exploit, because schema design and query optimization is tightly coupled with data partitioning.

Currently, we are working on an extension of this work: the evaluation presented in this article is limited in that we don't study the performance/cost scalability of the two systems (both scale out, and scale up), this is the main aspect we are set to evaluate in the progress of our work. Furthermore, we will study in more details the robustness and the impact of failures on the performance of both systems.

## Acknowledgements

The Authors would like to thank the anonymous reviewers for their helpful comments, that contributed to an improved version of our original submission. P. Michiardi and A. Barbuzzi were partially funded by the French ANR-VERSO project VIPEER. F. Riccato and A. Bär were supported by the Austrian Government and by the City of Vienna within the competence center program COMET.

## 6. REFERENCES

[1] http://hadoop.apache.org.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. SIGMOD Rec., 26(1):65–74, Mar. 1997. ACM ID: 248616.

[3] K. Cho et al. Traffic data repository at the wide project. In Proc. of USENIX ATC, 2000.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In Proc. of USENIX OSDI, 2004.

[5] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. Comm. of ACM, 35:85–98, 1992.

[6] M. Mellia, R. L. Cigno, and F. Neri. Measuring IP and TCP behavior on edge nodes with Tstat. Computer Networks, 47(1):1–21, 2005.

[7] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In Proc. of ACM SIGMOD, 2009.

[8] R. Stewart. Performance & Programming Comparison of JAQL, Hive, Pig and Java. Technical report, Heriot-Watt University, March 2010. Abstract of Results from MEng Dissertation.

[9] M. Stonebraker et al. MapReduce and parallel DBMSs: friends or foes? Comm. of ACM, 53:64–71, 2010.