JetScope: Reliable and Interactive Analytics at Cloud Scale

Eric Boutin, Paul Brett, Xiaoyu Chen, Jaliya Ekanayake
Tao Guan, Anna Korsun, Zhicheng Yin, Nan Zhang, Jingren Zhou
Microsoft

ABSTRACT

Interactive, reliable, and rich data analytics at cloud scale is a key capability to support low latency data exploration and experimentation over terabytes of data for a wide range of business scenarios. Besides the challenges in massive scalability and low latency distributed query processing, it is imperative to achieve all these requirements with effective fault tolerance and efficient recovery, as failures and fluctuations are the norm in such a distributed environment.

We present a cloud scale interactive query processing system, called JetScope, developed at Microsoft. The system has a SQL-like declarative scripting language and delivers massive scalability and high performance through advanced optimizations. In order to achieve low latency, the system leverages various access methods, optimizes delivering first rows, and maximizes network and scheduling efficiency. The system also provides a fine-grained fault tolerance mechanism which is able to efficiently detect and mitigate failures without significantly impacting the query latency and user experience. JetScope has been deployed to hundreds of servers in production at Microsoft, serving a few million queries every day.

1. INTRODUCTION

An increasing number of organizations rely on the results of massive data analytics for critical business decisions. As the data volumes expand dramatically, there is a need to scale out query processing and support efficient big data analytics over large clusters of commodity hardware. On the other hand, failures and fluctuations are the norm in such cloud scale clusters. It is critical for the system to be resilient to various system faults and efficiently recover from them with minimum performance impact. While the scale and complexity of data processing continues to grow, there is also an increasing demand to produce results in a real-time and interactive manner. Such low latency big data systems with fault tolerance greatly facilitates data exploration and fast experimentation, and enables a wide range of real-time business scenarios.

Parallel database systems have focused on latency optimizations, typically running on expensive high-end servers. However, when the data volumes to be stored and processed reach a point where clusters of hundreds or thousands of servers are required, parallel database solutions become prohibitively expensive. At such scale, many of the underlying assumptions of parallel database systems (e.g., fault tolerance) begin to break down, and the classical solutions are no longer viable without substantial extensions.

To address the scalability and reliability challenges, several batch processing systems were proposed, including MapReduce [8], Dryad [12], Hive [19], Scope [21], Pig [15], Stratosphere [3], Impala [13], etc., some of which offer high level SQL-like programming languages and conceptual data models. Such systems are designed to achieve great query throughput over both structured and unstructured datasets with scalability and fault tolerance. The queries usually process tens of terabytes of data or more in batch systems and can take up to hours or days to finish. It is difficult for these systems to achieve interactive query response times for several architectural reasons. For instance, the intermediate results are often materialized to disks, in order to provide fault tolerance and allow subsequent tasks to execute independently for scalability reasons. Such materialization costs are prohibitive for low latency queries. In addition, the underlying tasks are scheduled in phases, or batches. It is typical to have seconds of delay to start, track, and shut down tasks. These design choices are sufficient for batch workload, but fall short for low latency queries.

In this paper, we describe an interactive low latency query processing system at cloud scale, called JetScope, developed at Microsoft. The system has a SQL-like declarative scripting language with no explicit parallelism, while being amenable to efficient parallel execution on large clusters. Rich structural properties and access methods allow sophisticated query optimization and efficient query processing. An optimizer is responsible for converting scripts into efficient execution plans for the distributed computation engine. A physical execution plan consists of a directed acyclic graph (DAG) of tasks. Execution of the plan is orchestrated by a job manager that schedules tasks on available servers. In order to achieve low latency, the system utilizes a rich set of index structures and data structural properties and optimizes for pipelining results to users, avoiding any blocking operations. During execution, effective gang scheduling strategy is exploited to minimize task startup costs and in-

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

termediate results are streamed between tasks without hitting disks to reduce execution latency. The system scales well to hundreds of servers and is able to process datasets with terabytes of data in seconds.

Providing fault tolerance and failure recovery efficiently while maintaining low query latency at scale is particularly challenging. The approach of rerunning the entire query in case of failures is expensive and significantly increases the end-to-end query latency. JetScope implements a novel lightweight fault tolerance technique, which greatly minimizes wasted work and provides fine-grained failure recovery effectively. Specifically, JetScope tracks task execution progress continuously and is able to recover failed task with minimum recomputation. As a result, the overall impact from failure handling is greatly minimized without sacrificing query latency.

The JetScope system has been deployed to hundreds of servers in production at Microsoft. The system serves as the computation platform for various Microsoft services, targeted for large scale interactive data analysis, and executes a few million queries daily with a rapidly growing popularity. Experiments show that JetScope outperforms Impala and Hive by a few times on similar hardware. The system implements a scalable architecture which efficiently serves hundreds of queries with a variety of complexities concurrently. With fine grained fault tolerance mechanisms, JetScope is capable of recovering from server or rack failures more efficiently than other systems. Finally, JetScope evolves from Scope [21, 5], a cloud scale batch production system, which provides a natural unification between batch and interactive query processing. Albeit with different system tradeoffs and optimization goals, distributed batch and interactive query processing share lots of commonalities in declarative language, optimization, runtime, and scheduling.

The rest of this paper is structured as follows. In Section 2 we present the query language and data model in JetScope. We also describe rich access methods that facilitate query optimization and processing. In Section 3 we give an overview of the JetScope architecture. We present details of query compilation, optimization, runtime, and scheduling in order to achieve low query latency in the cluster in Section 4. In Section 5, we explain the lightweight task monitoring mechanism and present an efficient protocol to recover from failures and minimize latency impact. We share some interesting production experiences and explain how to effectively unify both batch and interactive query processing in Section 6. We present experimental results to compare JetScope with other systems, demonstrate efficiency of our novel failure handling strategy, and evaluate the system scalability in Section 7. Finally, we review related work in Section 8 and conclude in Section 9.

2. QUERY LANGUAGE & DATA REPRESENTATION

JetScope provides a powerful scripting language based on SQL. A script consists of a sequence of SQL commands, each of which takes one or more row sets as input, performs some operation on the data, and outputs a row set. Like relational databases, every row set has a well-defined schema that all its rows must adhere to. Users can name the output of a command using assignment and output can be consumed by subsequent commands simply by referring to it by name. Named inputs/outputs enable users to write scripts in multiple (small) steps, a style preferred by some programmers. Besides SQL queries, JetScope provides a seamless integration with C#, which allows users to call C# functions where needed.

```
Result =
    SELECT COUNT(*) AS count
    FROM store_sales,
         household_demographics,
         time_dim,
         store
    WHERE ss_sold_time_sk == time_dim.t_time_sk
      AND ss_hdemo_sk == household_demographics.hd_demo_sk
      AND ss_store_sk == s_store_sk
      AND time_dim.t_hour == @Hour
      AND time_dim.t_minute >= 30
      AND household_demographics.hd_dep_count == @Depcnt
      AND store.s_store_name == "ese"
    ORDER BY count;

OUTPUT Result TO CONSOLE;
```

Figure 1: TPC-DS Query 96 in JetScope.

Figure 1 shows TPCDS Query 96 in JetScope to illustrate an example. The last command requests the results to be streamed to the console window. The full description of JetScope programming language is outside the scope of this paper.

JetScope also supports a rich set of partitioning schemes and indexing methods, which enable sophisticated query optimization and facilitate query execution.

Partitioning Schemes. In JetScope, tables can be horizontally partitioned into tens of thousands of partitions. JetScope supports a variety of partitioning schemes, including hash and range partitioning on a single or composite keys. Based on the data volume and distribution, the system can choose the optimal number of partitions and their boundaries by means of sampling and calculating distributed histograms. Data in a partition is typically processed together (i.e., a partition represents a computation unit). A partition is comprised of one or several physical *extents*, which is the unit of storage in the underlying distributed file system. This approach allows the system to achieve effective replication and fault recovery through extents while providing computation efficiency through partitions.

Data Affinity. A partition can be processed efficiently when all its extents are stored close to each other. Unlike traditional parallel databases, JetScope does not require all extents of a partition to be stored on a single server which could lead to unbalanced storage across servers. Instead, the system attempts to store the extents close together by utilizing *store affinity.* Store affinity aims to achieve maximum data locality without sacrificing uniform data distribution. Every extent has an optional *affinity id*, and all extents with the same *affinity id* belong to an *affinity group.* The system treats store affinity as a placement hint and tries to place all the extents of an affinity group on the same server unless the server has already been overloaded. In this case, the extents are placed in the same rack. If the rack is also overloaded, the system then tries to place the extents in a close rack based on the network topology. Each partition of a table is assigned an *affinity id.* As extents are created

within the partition, they get assigned the same *affinity id*, so that they are stored close together.

Table References. The store affinity functionality can also be used to associate/affinitize the partitioning of an output table with that of a *referenced* table. This causes the output table to mirror the partitioning choices (i.e., partitioning function and number of partitions) of the referenced table. Additionally, each partition in the output table uses the *affinity id* of the corresponding partition in the referenced table. Therefore, two tables that are referenced not only are partitioned in the same way, but partitions are physically placed close to each other in the cluster. This layout significantly improves parallel join performance, as less data need to be transferred across the network.

Primary and Secondary Indexes. JetScope supports both primary and secondary indexes. Within each partition, a primary index is maintained over a single or composite keys. Optional secondary indexes can be created over a set of columns, potentially using a different partitioning scheme. The optimizer considers all the available indexes to generate the optimal execution plan.

Column Groups. JetScope supports both row and column store. For the row store, columns in a table are packed in a way that is similar to PAX [2] in an extent to improve computation and memory efficiency. To address scenarios that require processing just a few columns of a wide table, JetScope supports the notion of *column groups*, which contain vertical partitions of tables over user-defined subsets of columns.

3. ARCHITECTURAL OVERVIEW

JetScope is a multi-tenant and scalable interactive system delivering low latency query execution over tens of terabytes of data. It employs a scalable architecture, which scales up query processing over big data while supporting hundreds of parallel queries with a variety of complexities. In this section, we describe JetScope architecture and how a JetScope query is answered, from submission, compilation, optimization to execution and returning the results in a streaming fashion.

Figure 2 shows an overview of the JetScope system at scale. JetScope queries are submitted to the cluster portal either from users' development environment or various applications via ODBC APIs. Inside the computing cluster, JetScope comprises of three major layers: (i) front end, (ii) orchestration, and (iii) back end. The front end layer authenticates users and compiles the query. It hands the compiled query to the orchestration layer, which schedules and dispatches individual tasks to back end servers for execution. The data is read from a distributed file system, storing data across the cluster. Whenever possible, tasks are dispatched to the servers that are close to the input data in the network topology to take advantage of data locality. Once the query starts execution, the results are streamed back through the front end to the client application as soon as they are available. In order to support a large number of concurrent queries, the system automatically shards query compilation and scheduling among different instances of system components and provides efficient load balance among them. The capacity of the system can be dynamically adjusted by adding/removing servers in various functions.

3.1 Front End

The front end layer consists of a group of servers handling communication between the cluster and the clients. Each server runs a front end service, which performs authentication and provides interfaces for both job submission and cluster management, and a compiler service, which carries out job compilation and optimization.

A JetScope script goes through a series of transformations before it is sent for execution in the back end. Initially, the compiler parses the input script, unfolds views and macro directives, performs syntax and type checking, and resolves names. The result of this step is an annotated abstract syntax tree, which is passed to the query optimizer.

The JetScope optimizer is a cost-based transformation engine that generates efficient execution plans for input trees. It leverages existing work on relational query optimization and performs rich and non-trivial query rewritings that consider the input script in a holistic manner, takes account of data structural properties and available indexes, etc.

The compiler then generates code for each operator and combines a series of operators into an execution unit or *task*, which can be scheduled separately and executed by a single server. The system groups distinct types of tasks into separate *stages* to simplify job management. All the tasks in a stage perform the same computation, defined in the query plan, on a different partition of input data. The output of the compilation of a script thus consists of (i) a graph definition file that enumerates all stages and the data flow relationships among them, and (ii) the assembly itself, which contains the generated code. This package is sent to a job manager service (JMS) for execution. Throughout the execution of the query, the front end service serves as the gateway, which streams the resulting rows to the client application as soon as they become available.

3.2 Orchestration

The execution of a JetScope script is orchestrated by a job manager service (JMS), which is responsible for constructing the task graph using the definition sent by the compiler and scheduling work across available resources in the cluster. The JMS maintains the task graph and keeps track of the state and history of each task. To achieve low latency, the JMS uses special scheduling techniques to dispatch tasks in real-time, which is covered in detail in Section 4. The JMS also continuously monitors the status for each task, detects transient failures, and effectively recovers from them without rerunning the entire query. We discuss the fault tolerance strategies in Section 5.

To ensure all the servers run normally, a coordinator service maintains state information for each server and continuously tracks their health via a heartbeat mechanism. In case of server failures, the coordinator notifies every JMS so that no new tasks will be dispatched to the problematic servers. The coordinator also dynamically manages resources for each JMS. When new resources become available, for instance, new servers are added, each JMS is notified by the coordinator so that they can utilize their newly allocated resources for task execution. We discuss resource management in detail in Section 4.4.

3.3 Back End

There is a large group of processing servers in the back end, each of which runs a *Process Node* service (PN). The

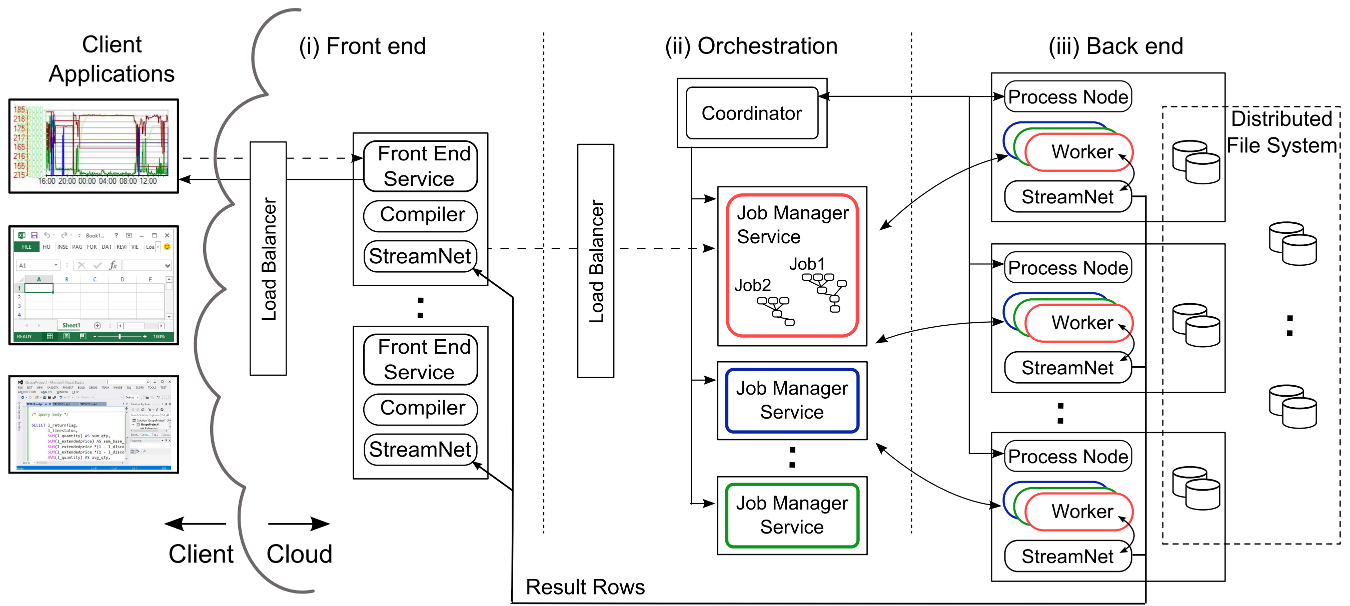


Figure 2: JetScope Architectural Overview.

PN service on each server is a proxy to manage *workers* that are ready to execute any assigned task promptly. At runtime, the JMS dispatches a task to a worker for execution. The worker is responsible to set up the required execution environment, including executing any required I/Os from the underlying distributed file system. Operators within a task are processed in a pipelined fashion, similar to a single-node database engine. As we shall describe in Section 4.2, the results of a task become immediately available for its consuming tasks.

4. LOW LATENCY OPTIMIZATIONS

A key aspect of an interactive analytic engine is to minimize query latency and deliver results to end users as soon as they become available. JetScope achieves low latency user experience by optimizing every component on the critical path of the query execution.

First, the compiler service is always ready to accept a new query and finishes compilation and optimization in a small fraction of a second. Second, the resulting DAG execution plan is sent to a long running JMS, which starts to construct graph and schedule tasks to back end servers immediately. Third, the system generates code for each operator and compiles it into machine code to avoid any interpretation costs at runtime. Finally, the results of a task are immediately available for the next task or routed back to end users without any delay. In addition, we describe several key techniques to avoid latency overhead in a distributed environment.

4.1 Query Optimization

JetScope has a cost-based query optimizer which leverages database optimization techniques [11] and reasons about structural properties such as partitioning, grouping and sorting properties; and various access methods holistically. The detailed description about the optimizer can be found in [22]. Further, to optimize for interactive experience, the query optimizer avoids blocking operators, such as sort-based operations, to favor hash-based operations so that any partial intermediate results can be pipelined to the next operation, possibly on a different server, without waiting for the complete results.

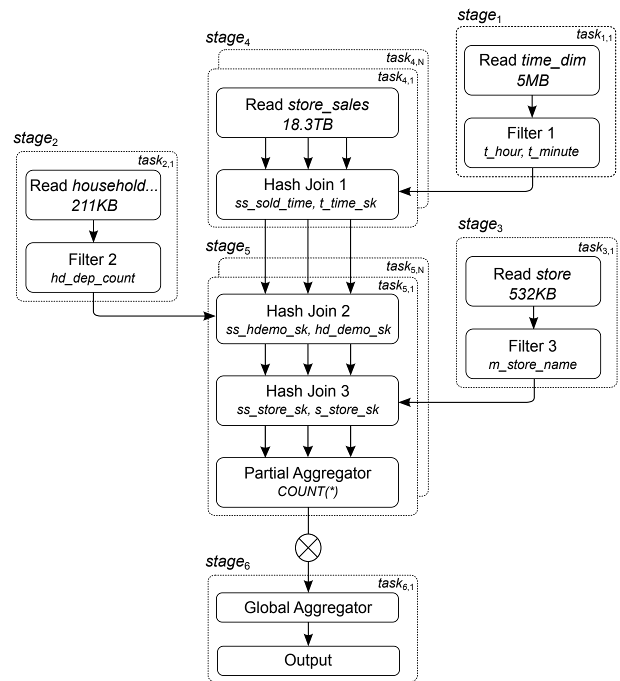


Figure 3: An Execution Plan for TPC-DS Query 96.

Figure 3 shows the final execution plan for TPC-DS Query 96 against a 30TB TPC-DS database by the optimizer in JetScope. There are 6 stages, some of which contain a group of tasks so that they can perform the same computation on a different partition of the input table in parallel. As

explained earlier, each task represents an execution unit and contains a series of physical operators, which are executed in a pipeline fashion. In this example, Query 96 joins the fact table `store_sales` with a few dimension tables `time_dim`, `household_demographics` and `store`. On one side, the fact table is over 18TB and the plan contains 1000 tasks in stage 4, each processing a small portion of the fact table. On the other side, the optimizer chooses to push necessary filters into individual dimension tables and broadcast the small result to the fact table for joins (in stage 4 and 5), respectively. Hash joins are chosen to optimize for fast results. Stage 6 aggregates all the partial results together.

To optimize for low latency, the optimizer splits the three hash joins into two separate stages, which can be executed on different servers concurrently. Such a plan has several benefits. First, each hash join needs to build an in-memory hash table on the corresponding join key. Running them on different servers allows better utilization of memory. Second, running them separately allows overlapping their computation and shorten the overall latency. Finally, the join 1 is far more expensive than the join 2 and 3 as the first join reduces the dataset significantly. By separating them into two separate stages, the plan is more reliable as it would be cheaper to recover from a server failure: the system only needs to rerun the affected task and their corresponding join operations, instead of recomputing all the three joins.

4.2 Network Communication

To minimize latency, JetScope also tries to pipeline intermediate results to subsequent tasks via network as much as possible. Such network communication between servers are bandwidth intensive and latency sensitive. Establishing a large number of TCP connections suffers from latency variations and occasional transient failures, especially in a busy network. In order to avoid re-establishing connections for every task, and to reduce the total number of TCP connections, a new service, called *StreamNet*, is used on every front end and back end servers to maintain long lived TCP connections between servers. A connection is shared by all tasks which need to communicate between a given server pair, regardless of their query of origin. Each task writes its intermediate rows to a memory buffer maintained by StreamNet. Then the rows immediately become available for other tasks to consume. The producer task doesn't wait for the consumer task to read the rows before processing others. Instead, it continues to execute and writes the result to a buffer, which is asynchronously spilled to disk when under memory pressure. This is necessary to prevent deadlocks which could otherwise happen when repartitioning data. This buffering is also required for the fault tolerance protocol, as described in Section 5.3.

4.3 Gang Scheduling

To further optimize the end-to-end latency, JetScope *gang schedules* all the tasks of a query at the same time. That is, a group of tasks are dispatched to different workers at once, potentially on different servers, so that they can execute concurrently. The approach has several benefits. First, as described earlier, the producer task always keeps the recent results in a memory buffer, before asynchronously writing them to disk. If both the producer and the consumer tasks run concurrently, the consumer task is able to read data immediately from the memory buffer when it becomes available, before it is spilled to disk, which greatly improves the query latency. Second, running both producer and consumer tasks concurrently enables tasks to overlap their computation, which effectively increases the parallelism of the query execution and reduces its overall latency. Third, unlike batch processing, there is no global synchronization point when executing the DAG of tasks and the first row latency can be significantly reduced.

Achieving efficient gang scheduling in a distributed environment is challenging and important, as any delay contributes to the overall query latency. A naive strategy is to incrementally acquire resources and hoard them until all resources are acquired. In our context, given that there are many JMSs which schedule tasks concurrently and independently, this could result in deadlocks. A typical strategy to resolve deadlocks is for each JMS to release all the hoarded resources after a timeout, then back off and retry. However, this strategy could introduce unpredictable latency at the query startup time. To solve these issues, in JetScope each JMS maintains and manages a pool of resources and employs an admission control mechanism based on instantaneous resource availability. When a query is received, the JMS evaluates how much resources are needed for the execution of the query, based on its execution plan, and if there is sufficient capacity in the pool, the query is admitted and immediately dispatched for execution, otherwise it is rejected. This mechanism avoids any deadlocks in resource acquisition and provides and instant answer to the client whether the query can be served or not at this moment. The client can perform additional retries using some strategy such as exponential backoff. Such user experience is much desirable in practice. The positive acknowledgment sent on query admission contains the metadata needed to read the stream of resulting rows as they become available.

4.4 Resource Management

In order to scale up to process hundreds of interactive queries concurrently, JetScope utilizes a group of JMSs, each of which is responsible for handling a group of queries, with load balanced globally among them. Since each JMS employs gang scheduling for each query to minimize its query execution latency, it becomes quite challenging to manage resources efficiently and effectively among different JMSs.

One approach is to utilize a centralized resource allocator which holds all the available resources. When a query is received, the JMS in charge of the query would ask the resource allocator for all the required resources. Once granted, the JMS uses the resource to schedule tasks of the query and returns the resources back to the coordinator when the query finishes. However, this approach has a few key drawbacks. First, the centralized resource allocator could become the scale-up bottleneck as the number of concurrent query requests increases. Second, the extra communication between the resource allocator and individual JMS can add undesirable overheads to the overall query latency. Finally, individual JMS no longer has the freedom to judiciously choose the ideal location for a task, based on data locality and failure impact considerations. Another approach is to have a shared-state scheduler [17], but such schedulers are designed to operate on a time scale too large for interactive query processing.

To meet all the performance requirements, in JetScope, the coordinator partitions all the resources so that each JMS

manages a disjoint portion. Each JMS admits and schedules queries concurrently and independently at a great speed. Each JMS also maintains some spare resources to handle unexpected failures, as described in Section 5. A load balancer monitors the availability of each JMS continuously and performs necessary load balancing among them. As described in Section 3, the coordinator can dynamically adjust resource allocations among different JMSs as needed. The coordinator also monitors server health and notifies individual JMS promptly of problematic servers in case of failures.

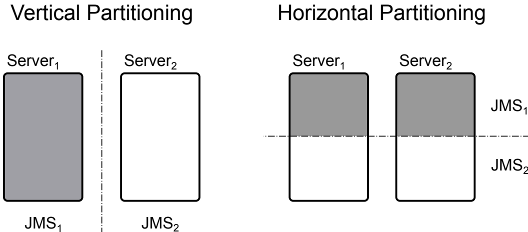


Figure 4: Resource Partitioning.

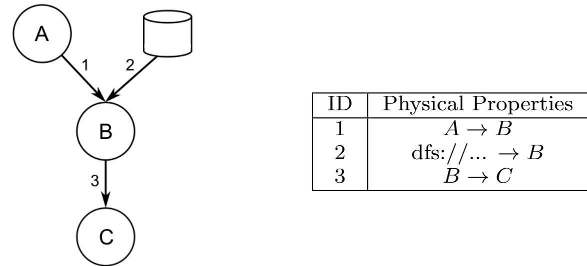
The coordinator supports several policies to distribute resources among different JMSs. One policy is to *vertically* partition resources at a server level where different JMSs are in charge of different set of servers. While this approach provides excellent performance isolation, it could result in suboptimal server utilization as each JMS cannot saturate all the assigned resources due to the short lived nature of interactive workload. In JetScope, the coordinator *horizontally* partitions all the resources, also called *resource stripping*. As shown in Figure 4, each server divides its own resources and distribute them among several JMSs. Not every JMS can fully utilize its assigned resources at all the times, therefore, overlapping the resources owned by individual JMS helps achieve balanced workload across all the servers. In practice, we observe the load on each server is well balanced and each server is decently utilized.

5. FAULT TOLERANCE

Failures and fluctuations are typical in cloud-scale distributed systems. Detecting failures and providing effective failure recovery in a timely manner is crucial for low latency query processing systems. The naive approach of rerunning queries in case of any failure wastes the existing computation progress and adds considerable latency as the query needs to be rerun from scratch. The drawback becomes even more obvious as the number of tasks and the amount of required resource for the query execution increase. Specifically, the probability that any of the involved servers fails during query execution grows significantly as the number of servers involved increases. As a result, big queries are likely to suffer from random failures repeatedly before any successful execution or, even worse, never succeed. Therefore, having a finer-grained fault tolerance control with great efficiency is essential. JetScope achieves fault tolerance using a real-time monitoring mechanism to identify failures and a novel fine-grained recovery approach which only performs the necessary recomputations greatly minimizing the latency impact.

5.1 Tasks and Channels

In JetScope, a task's I/O are performed through an abstraction called a *channel*, which represents a directed I/O stream. Before digging further into the fault tolerance protocol, we cover the channel abstraction in details.



ID	Physical Properties
1	$A \rightarrow B$
2	dfs://... $\rightarrow B$
3	$B \rightarrow C$

(a) Task Graph (b) Mapping Table

Figure 5: A Channel Abstraction Example.

Every task has both input and output channels, corresponding to the input and output I/O streams, respectively. Each channel has a unique *logical* ID and *physical* properties which describe the implementation of the represented I/O stream (e.g. how to read from it and write to it). While the logical ID of a channel is constant throughout a query, the physical properties of a channel can be changed during the query execution. The mapping of channel logical ID to physical properties is maintained by the JMS in a *mapping table*. Multiple physical implementations of channels exist. For example, a channel can be used to read from a distributed file system, or from another task through StreamNet.

Data written to channels is required to be deterministic. Channels can be read at any position. While a channel is *active* (i.e. the producing task is still running), any read request for data that has not yet been generated (e.g, to read data at a non-existing position) is blocked until the producing task generates the required data. A channel is *finalized* when the producing task completes. Once a channel is finalized, any pending read requests for a position that is beyond the length of the finalized channel will fail.

Figure 5 illustrates a task B and its channels. In the example, the task has two input channels. As described in the mapping table, one input channel connects the task to an upstream task, while the other input channel connects the task to a table partition stored in a distributed file system. The task has one output channel, which is used to send data to a downstream task. The physical properties of a channel can be changed while a task is running. For instance, suppose that task A has another instance, called A', running concurrently as a duplicated execution, they produce the same deterministic results. Depending on the physical properties of its input channel, task B can read from either A or A' with a specific offset, without affecting the correctness of the results. In fact, task B can issue the same request to both A and A' and immediately consume the data if one returns. Such a powerful abstraction also simplifies failure recovery described next.

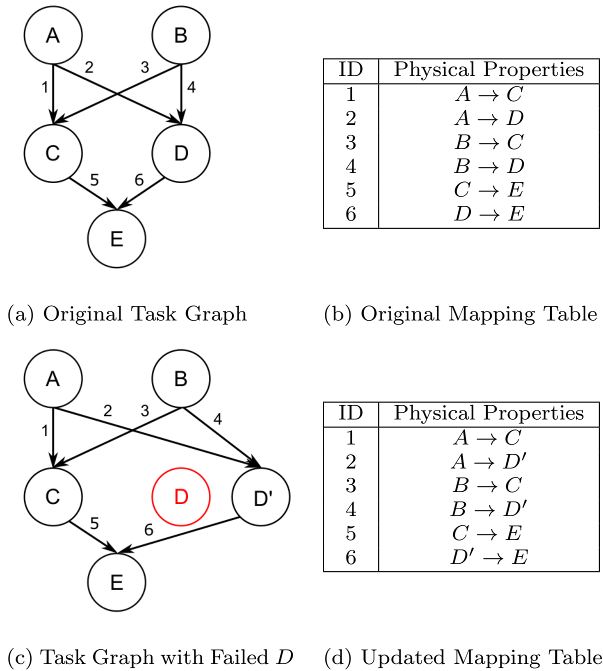
5.2 Real-Time Fault Detection

Timely detection of failures in a distributed environment is challenging, as there are so many factors that may contribute to different kinds of failures. Some failures are espe-

cially difficult to detect or are only detected after they make negative latency impact. For instance, network partitions or power outages, which do not usually result in an immediately observable query level failures. Instead, they typically result in low level system events, such as network timeouts or TCP connection failures, which are only detected after the query latency was impacted.

JetScope employs two lightweight mechanisms to detect and identify failures promptly for each task. First, JetScope uses a frequent heartbeat communication between the JMS and each worker running a task. The JMS keeps track of the last time it received an update from a task. When a task missed enough updates, the JMS considers the task failed and restarts the task in another server. This mechanism allows the JMS to quickly detect failures affecting running queries and take immediate actions.

Second, the coordinator maintains communication with each back end server. When the coordinator detects a server failure due to lost communication, it notifies every JMS, which then considers all tasks on the server failed and triggers recovery. The coordinator then requests a replacement back end server in order to avoid a capacity reduction. The second mechanism notifies each JMS of server failures, even if some JMS had no task scheduled on the server. It prevents future tasks from being dispatched to the problematic servers, which would otherwise results in query processing delays.



ID	Physical Properties
1	$A \to C$
2	$A \to D$
3	$B \to C$
4	$B \to D$
5	$C \to E$
6	$D \to E$

(b) Original Mapping Table

ID	Physical Properties
1	$A \to C$
2	$A \to D'$
3	$B \to C$
4	$B \to D'$
5	$C \to E$
6	$D' \to E$

(d) Updated Mapping Table

Figure 6: A Failure Recovery Example.

5.3 Fine-Grained Failure Recovery

JetScope streams intermediate results between dependent tasks, allowing downstream tasks to consume rows when they are available. Although this helps reduce the query latency, it introduces challenges when failures occur in the middle of communications. JetScope employs a *fine-grained* fault recovery mechanism, which enables recovery with minimal latency impact by only recomputing the failed tasks.

Figure 6 illustrates this mechanism. In this example, we consider a DAG containing 5 tasks, shown in Figure 6(a). Each arrow between two tasks represents an I/O channel, annotated with the logical ID of the channel. Figure 6(b) shows the original mapping table.

Figure 6(c) shows the modified task graph after task D fails. A new instance of task D, denoted as D', is started on a different server. The task D' inherits its logical channels (2, 4, and 6) from task D. The mapping table is updated accordingly, as shown in Figure 6(d). Specifically, channels 2 and 4 are now consumed by D' and channel 6 is produced by D'. The updated mapping is then sent to the tasks via the heartbeat message. Upon receiving this information, the affected tasks (A, B, and E) update the physical properties of their input and output channels and resume execution.

Specifically, task D' restarts its computation from the beginning by consuming outputs from A and B via channels 2 and 4. Task A and B do not need to restart. Instead, they continue to process and write data into their output channels, unaffected by the failure, throughout the recovery. The read requests issued by task E on channel 6 will simply block until D' catches up and writes at the position requested. The high-level protocol is described in Algorithm 1.

Algorithm 1: Failure Detection and Recovery

foreach *task* T_i *in the query* **do**
  if T_i *missed* K *updates* **then**
    `// ` T_i ` is considered failed,`
    `// start a replacement task`
    T_i' = `StartReplacementTask` (T_i);
    `// Update the mapping table`
    foreach C_j *in* `InputChannels` *(T_i)* **do**
      `// Connect ` C_j ` to the new consumer`
      p = `Producer` (C_j) ;
      `UpdateChannelMap` (C_j, $p \to T_i'$);
    foreach C_j *in* `OutputChannels` *(T_i)* **do**
      `// Connect ` C_j ` to the new producer`
      c = `Consumer` (C_j) ;
      `UpdateChannelMap` (C_j, $T_i' \to c$);

5.4 Reliable Task Scheduling

The JMS in JetScope employs a cost-based scheduler to improve task scheduling quality and optimize efficiency [5]. The scheduler considers a wide range of factors, such as data locality, server health, server utilization, etc. in a combined cost model, which is used to choose the ideal server to schedule a task. For each task, a cost is assigned to a candidate server, based on the expected execution time of the task if scheduled to the server, the probability that the server fails during the query execution, and the cost of the failure (e.g. the cost of rerunning the task). The scheduler is originally designed for batch queries but the same scheduling strategy applies to interactive queries in JetScope. The detailed description of the scheduler can be found in [5].

An interesting challenge when scheduling interactive queries is handling the *correlated failures* and reducing their impact on the query latency. For instance, assume task t_{k-1} is a parent of task t_k, that is, t_k consumes outputs from t_{k-1}.

If both tasks are scheduled to the same server or rack, both tasks would fail if the server or the rack failed. In this case, the recovery of the query is slowed down, because not only both tasks need to rerun but also t'_k has to wait for t'_{k-1} to repopulate the output channel. Such correlated failure are common in practice and could have severe impact on the system reliability and query latency. JetScope incorporates such considerations into the cost model. The cost of a server failure now needs to consider all the dependent tasks that get scheduled to the same server. As a result, the scheduler judiciously considers scheduling two expensive and dependent tasks in different failure domains to avoid such correlated failures in a cost-based manner.

6. DISCUSSION

JetScope is designed to be an interactive computing service, running 24/7 without any down time. It is important to handle system deployments and maintenance gracefully without impacting users. We describe key architectural aspects of JetScope that allows online system deployment and maintenance in Section 6.1. JetScope evolves from Scope [21], a distributed batch processing system. We discuss an important unification of batch and interactive query processing in Section 6.2.

6.1 Online Deployment and Maintenance

JetScope is designed to support multiple versions of the system to serve queries concurrently. The system can run different versions of front end services, JMSs and their workers side by side. Upon query submission, the system assigns the query a version tag, which is used to route the query to the system of the corresponding version for further processing. Such capability is crucial for online deployment, production flighting, and verification. During those events, the system temporarily stops routing queries, via load balancers, to a set of system components of a particular version, waits for them to drain current computation, and takes them down for upgrade or deployment before routing new queries to them. During the process, another version of the system continues to operate, with minimal impact on users.

Cluster wide maintenance, e.g., to apply OS patches, occur regularly in data clusters. Similarly, JetScope applies maintenance *progressively* to minimize their impact to query processing. Specifically, the coordinator first receives notification that certain servers are scheduled for maintenance, and then notified each JMS to stop sending new tasks to the affected servers. The system waits for the current tasks to finish before taking them down for maintenance. Finally, the coordinator picks new servers to compensate the lost capacity and notifies JMSs. The rest of the system continue to function during this process.

6.2 Unification of Batch and Interactive Processing

Batch processing systems typically handle massive datasets, optimize for throughput, and can take hours or days to finish processing a query, while interactive processing optimizes for low latency and process queries in seconds or minutes. The two systems compliment each others and both are valuable in the big data computation eco-system. It is therefore important to unify batch and interactive processing effectively and support both in a shared environment. JetScope and Scope provide a natural unification, as many system components are shared between batch and interactive processing but optimize different aspects in execution, scheduling, and fault tolerance. The two systems are deployed to the same computing clusters with tens of thousands of servers, each with its own allocated resources. The two systems share the same underlying distributed file system, storing data across all the servers in the cluster. Such a design allows users to leverage both systems for different scenarios without copying data from one system to the other. In fact, it is typical to use the batch processing system to process the raw data and construct tables, which is time-consuming, and then leverage the interactive processing system for fast analytics and exploration. In addition, the unification allows users to write queries using the same declarative language. Performing automatic selection of the optimal execution strategy, depending on the data volume, required resources, etc. is part of our future work.

7. EVALUATION AT SCALE

We perform detailed experiments on JetScope to evaluate its performance and scalability under various circumstances. More specifically, we focus on answering the following questions: (i) How does JetScope perform in a real production environment? (Section 7.1) (ii) How does JetScope perform with complex queries? How does it compare with the performance of other systems? (Section 7.2) (iii) How do server and rack failures affect the query performance with JetScope's fine-grained fault recovery? How does it compare with a coarse-grained fault recovery strategy? Does JetScope's task placement optimization help? (Section 7.3) (iv) How does JetScope scale? (Section 7.4)

7.1 JetScope in Production

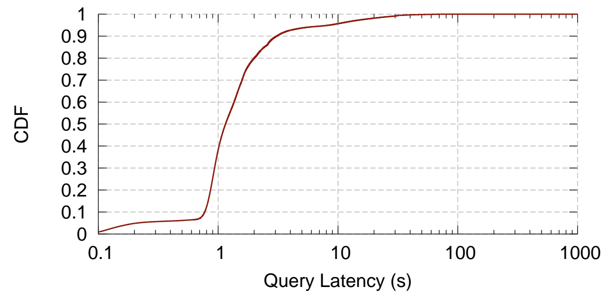


Figure 7: Query Latency on Production Clusters.

JetScope has been deployed to hundreds of servers in production at Microsoft, serving millions of queries against a wide range of big datasets on a daily basis. Figure 7 illustrates the distribution of query latencies observed. 40% of the queries complete in less than a second. Most of these queries benefit significantly from effective indexing and are often generated by reporting tools, built for common analytical scenarios. The rest of the workload consists of a wide range of ad-hoc data exploration and interactive experimentation queries. As the system supports both batch and interactive processing in a shared environment, users

run scaled down versions of more complex experiments using JetScope to validate their assumptions before launching full scale experiments that could run for many hours in the batch system.

7.2 Query Performance

JetScope is capable of running various complex SQL queries with excellent performance. However, it is difficult to compare the performance numbers among different distributed computation systems, as the underlying computing clusters might have different hardware configuration, which have a profound impact on the system performance. Recent work published detailed performance numbers about Hive and Impala running on a 1TB TPC-H database, as well as the hardware configuration used [10]. We configure a test cluster with a similar hardware configuration so as to roughly compare JetScope performance numbers with those for Hive and Impala. In particular, the test cluster has 21 servers as in the Hive and Impala experiments and each server has comparable specifications.

	Hive & Impala [10]	JetScope
Server Count	21	21
Server Configuration		
OS	Ubuntu 12.04	Windows Server 2012
Memory	96GB	128GB
Network	10 gigabit	10 gigabit
CPU Sockets	2	2
CPU Cores	12	16
CPU Specification		
Brand	Intel Xeon	Intel Xeon
Clock	2.2GHz	2.1GHz
Cores/Socket	6	8
I/O System		
Disk Count	200	180
Disk RPM	7k	7k

Table 1: Cluster Configuration.

It is worthwhile pointing out the difference in the cluster configuration. For Hive and Impala experiments, each server has 10 direct-attached hard drives. In JetScope, each server has 3 direct-attached data drives. To have a similar configuration, we have data stored in 60 servers with 180 disks in total. This is suboptimal for JetScope as the data is distributed among different servers and the system no longer benefits from the data locality. On the other side, each server in JetScope clusters has slightly more cores and memory with slightly lower CPU clock speed. Despite the difference, we believe the cluster configuration is close enough to provide a rough comparison. To avoid diluting the performance numbers by caching data in memory, we disable file cache for JetScope.

Table 2 shows the latency of all 22 TPC-H queries over a 1TB database for Hive-Tez with data in the ORC Snappy format, Impala with data in the Parquet format, and JetScope with data stored natively in tables. The Hive and Impala numbers reported from a previous study [10], each representing the best performing setup. JetScope performs significantly better than Hive and Impala, and completes the benchmark 4 times faster than Impala, with some queries running as much as 10 times faster. Both JetScope and Impala perform faster than Hive. Interestingly, there are

Query	Hive-Tez [10]	Impala [10]	JetScope	Speedup over Impala
1	172	25	19	1.28
2	111	29	21	1.41
3	280	122	42	2.92
4	214	135	36	3.72
5	409	189	44	4.27
6	81	6	16	0.37
7	589	179	99	1.81
8	369	140	54	2.61
9	1692	Failed	119	n/a
10	224	59	41	1.44
11	134	22	14	1.56
12	184	38	43	0.88
13	156	123	40	3.04
14	120	13	26	0.5
15	156	21	34	0.62
16	133	44	9	4.85
17	724	385	91	4.23
18	672	185	68	2.7
19	569	703	44	16.07
20	175	195	36	5.39
21	Failed	854	72	11.94
22	130	29	23	1.27
Sum	7294+	3496+	871	4.01

Table 2: TPC-H Query Latency over 1TB (seconds).

four queries Q6, Q12, Q14, and Q15 that run faster in Impala. The dominating cost of those queries is to read the big `lineitem` table and they could have benefited from the fact that data is partially in file cache, as done in the Hive and Impala experiments [10]. JetScope query runs do not use file cache.

Our investigation shows that JetScope produces better execution plans and utilizes a more efficient runtime engine to achieve low latency with scalability. One such example is Q19, in which JetScope is 16 times faster. Q19 performs a join between `lineitem` and `part` tables and apply a filtering predicate. The JetScope optimizer creates implied predicates and pushes them down to the table scanner. The selectivity of these predicates is very low — 2% for `lineitem` table and 0.5% for `part` table, and only a small fraction of rows reaches the join. Impala, on the other hand, does not pushdown predicates [10] and thus the join stage processes much more data.

7.3 Fault Tolerance

We evaluate the effectiveness of Scope fault tolerance strategy using 495 servers in one production cluster. Specifically, we run TPC-DS Q96 (shown in Section 2) against a 30TB TPC-DS database, manually inject different kinds of failures, and measure how fast the system recovers and the impact on the overall query latency.

There are two common types of failures in data clusters: (i) server level failures, such as disk failures, OS crashes, etc.; and (ii) infrastructure failure, such as power, cooling or network failure. The second type of failure is not rare in cloud-scale data clusters because the power and network infrastructures are often not redundant at the rack level. When it happens, all the servers in a rack, typically between 30 and 40 servers, become unavailable.

We first measure the query latency when there are no failures as the baseline and normalize it to 100. We then rerun the query, injecting a server or rack failure at a specific timing (t_{fault}), measure the final end-to-end query latency, and

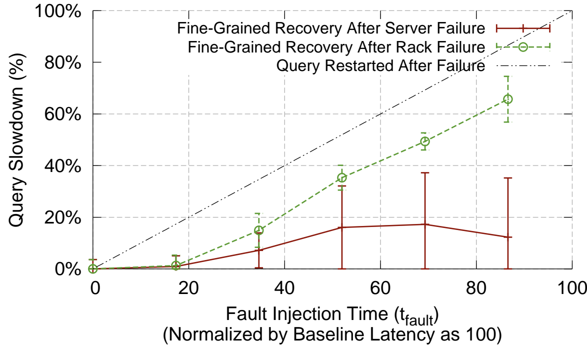


Figure 8: Impact of Failures on Latency.

calculate their slowdown (in percentage) compared to the baseline, respectively. Each experiment is repeated 5 times. We report the average and standard deviation of the query slowdown in Figure 8. We further compare with a coarse-grained fault tolerance strategy, where the entire query is reran in case of failures. Naturally, the coarse grained fault tolerance strategy has a slowdown which is linear with the time at which a failure is injected. For instance, if a failure is injected at $t_{fault} = 50$, the query will be restarted and will complete at $t = 150$, leading to a 50% slowdown.

As shown in Figure 8, each failure does slow down query execution as it takes time to recover. A rack failure is more expensive to recover than a single server failure, as it affects a group of tasks and takes more effort to recover. Single server failures have a big variance because the impact varies depending precisely on which tasks are affected. JetScope's fine-grained failure recovery performs significantly better than the approach of rerunning the entire query, regardless of the failure type. For example, in the worst case, when a rack failure is injected at $t_{fault} = 85$, the query completes with 63% slowdown (that is, an end-to-end latency $l = 163$), while restarting the query at time $t_{fault} = 85$ would result in a latency $l = 185$ (85% slowdown). A surprising case is when a rack failure is injected at $t_{fault} = 20$, and no additional slowdown is observed. Upon close inspection of the data, we observed that the failures were not on the critical path.

7.4 Scalability

Next, we evaluate the scalability of the JetScope system by running TPC-H Q1 against a 1TB database with different degrees of parallelism. The execution plan is relatively straightforward. Tasks are assigned different partitions of the `lineitem` table and perform local aggregation independently. Their results are aggregated by a separate task, which performs the final global aggregation. By changing the number of assigned tasks, the degree of parallelism increases, which improves the overall latency. Figure 9 shows that JetScope achieves a near-ideal speedup from 16 tasks to 180 tasks for this query.

Finally, we evaluate how JetScope scales as the amount of the processed data increases. Specifically, we use several TPC-H queries with increasing complexities and compare their latency using 20 servers against a 1TB database with that of using 200 server against a 10TB database. As illustrated in Figure 10, the latency stays almost constant,

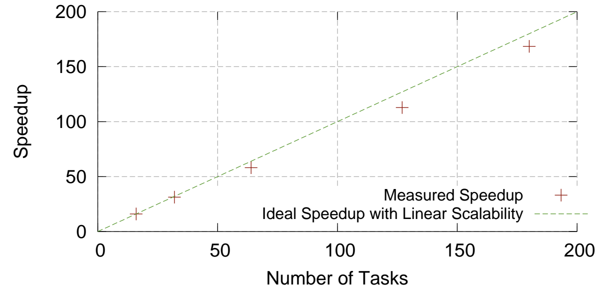


Figure 9: Latency with Different Numbers of Tasks.

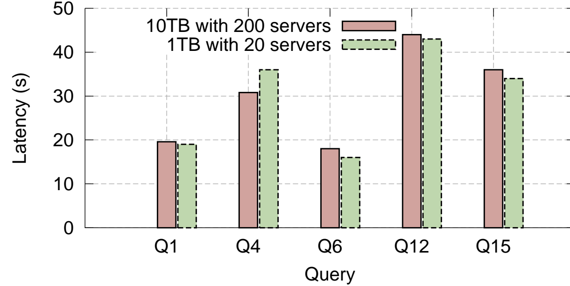


Figure 10: JetScope Scalability.

despite the 10x increase in scale. These results demonstrate the excellent scalability of the system.

8. RELATED WORK

Cloud-scale batch processing systems like Map-Reduce [8], Scope [21], Pig [15] and Hive [19] are designed for very large scale data processing where a single job can process a few petabytes of data. One key design choice for such systems is to mainly optimize the system throughput. Due to its massive scale, it becomes critical to deal with various system fluctuations and failures. For instance, those systems typically trade latency for reliability and scalability by materializing intermediate results to disks.

By contrast, interactive query processing systems process gigabytes to tens of terabytes of data and strives to achieve low latency execution. MapReduce Online [7] is a data processing system which streams data through operators to leverage pipelining and reduce the materialization cost. Scuba [1] achieves low latency by storing temporal data in memory and executes queries by aggregating data through dynamic execution trees. Failures are handled by approximating results. Dremel [14] executes queries by leveraging dynamic aggregation trees to read, process, and aggregate data stored in columnar format. F1 [18] supports complex communication graphs over a distributed transactional store, and uses coarse-grained fault tolerance. F1 also supports transactional updates to the data. Spark [6, 4] and Shark [20] systems achieve low latency by aggressively leveraging RDD and provides fault tolerance through lineage recomputation. Impala [9] is an MPP SQL query engine which extensively leverages effective code generation, a columnar data format, and an efficient I/O subsystem which short-circuits the distributed file system to improve I/O performance. JetScope leverages database techniques and sup-

ports a wide range of complex queries with sophisticated query optimization. In addition, JetScope has a highly scalable architecture to process hundreds of queries concurrently in a truly multi-tenant cloud environment. Special techniques in query optimization, network communication, and scheduling are utilized to further optimize for query latency. The fine-grained fault tolerance mechanism enables the system to detect and recover from failures promptly and minimize any latency impact.

High performance scheduling with good quality is essential when scheduling large scale interactive queries in a cloud environment. Omega [17] is a shared-state resource manager which allows schedulers to make independent decisions, and support transactional acquisition of resources. Sparrow [16] is a scheduler designed for low latency workloads. It operates using batch sampling of the queue length of servers and schedule tasks on the servers with minimal queue length. Locality is implemented as a scheduling constraint. Apollo [5] is a cost-based scheduler which schedules tasks by judiciously trading off various factors such as load and locality to minimize task runtime using an estimation model. JetScope improves scheduling efficiency further by gang scheduling tasks to allow full pipelining of the task execution. It also considers correlated failures and their impact on the query latency to make the query execution less vulnerable to any random system failures.

9. CONCLUSION

We present a cloud scale interactive query processing system, called JetScope, developed at Microsoft. The system combines the benefits of parallel database systems and distributed low latency query processing engine. Specifically, it has a SQL-like declarative scripting language and delivers massive scalability and high performance through advanced optimizations. In order to achieve low latency in query processing, JetScope leverages various access methods, optimizes first rows delivery latency, and maximizes network and scheduling efficiency. To handle failures that are common in cloud scale computing clusters, JetScope provides a fine-grained fault tolerance mechanism which is able to efficiently detect and mitigate failures by only recomputing part of the query and minimizing any latency impact. JetScope has been deployed to hundreds of servers in production at Microsoft, serving a few million queries with a variety of complexities daily in interactive speeds.

10. ACKNOWLEDGMENTS

We would like to thank the following people for their contributions to the JetScope system: Nico Bruno, Bill Fan, Haochuan Fan, Rashim Gupta, Pavel Iakovenko, YongChul Kwon, and Martin Neupauer. We would also like to thank the members of the Microsoft Scope team for their contributions to the Scope distributed computation eco-system, of which JetScope serves as the interactive computation component. Finally, we are grateful to all the members of the Microsoft Big Data team for the support and collaboration.

11. REFERENCES

[1] L. Abraham, J. Allen, O. Barykin, V. R. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into Data at Facebook. *PVLDB*, 6(11):1057–1067, 2013.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[5] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, Oct. 2014. USENIX Association.

[6] Y. Chen, S. Alspaugh, and R. H. Katz. Interactive Query Processing in Big Data Systems: A Cross Industry Study of MapReduce Workloads. Technical Report UCB/EECS-2012-37, EECS Department, University of California, Berkeley, Apr 2012.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] M. K. et.al. Impala: A Modern, Open-Source SQL Engine for Hadoop . In *Proceedings of the Conference on Innovative Data Systems Research 2015*, CIDR '15, 2015.

[10] A. Floratou, U. F. Minhas, and F. Ozcan. SQL-on-Hadoop:Full Circle Back to Shared-Nothing Database Architectures. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[11] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[13] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang,

N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[14] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[16] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[17] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[18] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Little?eld, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A Distributed SQL Database That Scales. In *VLDB*, 2013.

[19] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of The Vldb Endowment*, 2:1626–1629, 2009.

[20] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[21] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel databases meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.

[22] J. Zhou, P. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 1060–1071, 2010.