Bubble Execution: Resource-aware Reliable Analytics at Cloud Scale

Zhicheng Yin, Jin Sun, Ming Li, Jaliya Ekanayake

Haibo Lin, Marc Friedman, José A. Blakeley[*] , Clemens Szyperski
Microsoft

Nikhil R. Devanur
Microsoft Research

ABSTRACT

Enabling interactive data exploration at cloud scale requires minimizing end-to-end query execution latency, while guaranteeing fault tolerance, and query execution under resource-constraints. Typically, such a query execution involves orchestrating the execution of hundreds or thousands of related tasks on cloud scale clusters. Without any resource constraints, all query tasks can be scheduled to execute simultaneously (gang scheduling) while connected tasks stream data between them. When the data size referenced by a query increases, gang scheduling may be resource-wasteful or un-satisfiable with a limited, per-query resource budget. This paper introduces Bubble Execution, a new query processing framework for interactive workloads at cloud scale, that balances cost-based query optimization, fault tolerance, optimal resource management, and execution orchestration. Bubble execution involves dividing a query execution graph into a collection of query sub-graphs (bubbles), and scheduling them within a per-query resource budget. The query operators (tasks) inside a bubble stream data between them while fault tolerance is handled by persisting temporary results at bubble boundaries. Our implementation enhances our JetScope service, for interactive workloads, deployed in production clusters at Microsoft. Experiments with TPC-H queries show that bubble execution can reduce resource usage significantly in the presence of failures while maintaining performance competitive with gang execution.

PVLDB Reference Format:
Zhicheng Yin, Jin Sun, Ming Li, Jaliya Ekanayake, Haibo Lin, Marc Friedman, José A. Blakeley, Clemens Szyperski, Nikhil R. Devanur. Bubble Execution: Resource-aware Reliable Analytics at Cloud Scale. *PVLDB*, 11(1): xxxx-yyyy, 2017.
DOI: https://doi.org/TBD

1. INTRODUCTION

Analyzing hundreds of terabytes of data on large clusters of commodity hardware is becoming the norm in today's big data exploration and analytic scenarios. These clusters, which are often configured as multi-tenant systems to run thousands of concurrent queries per hour, impose limits or constrains on the amount of resources available per query. At the same time, failures and workload fluctuations are common in such cloud scale clusters. While the scale and complexity of data processing continues to grow, business requirements to reduce "time-to-insight" demand a significant reduction in end-to-end query execution latency. A low-latency big data system that is both resource-aware and fault tolerant, greatly facilitates fast data exploration, data analysis, and a wide range of real-time business scenarios.

Existing systems use many optimization techniques to achieve low-latency query execution. They use optimized record formats, such as column-store [21] along with algorithmic query execution innovations such as vectorized expression evaluation [2] to provide very low query execution latencies. Running such engines on expensive high-end nodes, e.g. parallel databases, gives very low query latencies. However, when the data size grows to a point where hundreds or thousands of compute nodes are required, parallel database solutions become unfeasible or prohibitively expensive and assumptions of parallel database systems (e.g., no fault tolerance) break down.

Recent interactive query engines address the requirements of ad-hoc real-time analytics, such as Dremel [24], Tenzing [22], JetScope [3], Spark [1], and Presto [10]. Most of them offer high-level SQL-based query languages and conceptual data models, with support for data loading, and import/export functionality to connect to cloud storage or data warehousing systems. They employ optimized file formats, in-memory processing, and direct inter-operator data transfer techniques to achieve low latencies while fault tolerance is achieved via check-pointing or lineage. However, some of these techniques have implicit limitations. For example, our previous work on JetScope uses operator-to-operator pipe channels for data transfers and hence requires full gang semantics [11] when scheduling. Similarly, Spark provides fault tolerance through lineage which requires persistence among wide dependencies. On the other hand, gang execution enables the whole query to be running in a streamline manner, minimizing latencies. Gang execution, however, becomes expensive for complex queries involving thousands of

[*]Deceased January 7, 2018.

Proceedings of the VLDB Endowment, Vol. 11, No. 1
Copyright 2017 VLDB Endowment 2150-8097/17/09... $ 10.00.
DOI: https://doi.org/TBD

operators, by holding compute resources longer than needed for the execution. Furthermore, to provide fault tolerance, the intermediate data needs to be persisted at each corresponding execution operator, which increases latency.

This paper introduces Bubble Execution, a new query processing framework for interactive workload at cloud scale, that balances cost-based query optimization, fault tolerance, optimal resource management, and execution orchestration. Bubble execution can be adapted to general big data analytic engines. Our specific implementation of bubble execution builds on our previous JetScope service, for interactive workload, deployed in production clusters at Microsoft. JetScope supports the Scope language, a SQL-based declarative scripting language with no explicit parallelism, while being amenable to efficient parallel execution on large clusters. A cost-based query optimizer is responsible for converting scripts into efficient, physical, distributed query execution plans represented by a directed acyclic graph (DAG) of query operators (tasks). The execution of a plan is orchestrated by a job manager that schedules tasks on available compute resources. With bubble execution, the optimizer revisits the plan graph to aggregate or partition tasks into sub-graphs, called bubbles, each of which will satisfy the per-query resource constraints. Next, the job manager schedules tasks as gangs at bubble boundaries. Multiple bubbles can execute in parallel provided the sum of bubble sizes does not exceed the per-query resource quota. The intermediate results can be materialized at bubble boundaries providing an implicit checkpoint for the scheduler to recover when some tasks fail.

Providing fault tolerance efficiently while maintaining low query latency at scale is particularly challenging. The simple approach of rerunning the entire query in case of a failure is expensive and significantly increases the end-to-end query latency. Bubble execution implements a lightweight fault tolerance technique by leveraging different types of communication channels between tasks. Intermediate results are in-memory-streamed within a bubble, without hitting disks, to reduce execution latency. At bubble boundaries, temporary results are persisted to disk so that failure can be recovered from the beginning of a bubble rather than the beginning of the query. Thus, the overall impact of failure recovery is greatly minimized without sacrificing query latency.

Bubble execution has been deployed to hundreds of compute nodes in production on the Microsoft Cosmos service. Bubble execution serves as the distributed query processing platform for various services, targeted for large-scale interactive data analysis. It enables queries that were previously too expensive to run in JetScope and provides users with the ability to control the resources usage of their query pipelines. Bubble execution leverages the scalable architecture of JetScope which efficiently serves tens of hundreds of concurrent queries per hour with a variety of complexities.

The Microsoft Cosmos service runs a variety of workloads including batch data preparation, interactive, streaming and machine learning. We evaluated bubble execution against batch and interactive (gang) query execution and scheduling policies using queries based on the TPC-H benchmark. In batch mode, each task is scheduled independently and can be recovered independently. In gang mode, all query tasks are scheduled simultaneously and often employ all or nothing recovery semantics. Alternatively, bubble execution provides a combined semantics where tasks inside a bubble are scheduled as a gang, and bubbles can be scheduled independently from each other. Experiments show that bubble execution outperforms batch execution and achieves comparable performance to gang execution using fewer resources. For example, bubble execution could reduce 50% of resources usage with 25% of slowdown compared to gang execution, and half of the test cases achieve similar performance. This gives system administrators configuration flexibility to trade-off between latency and concurrency, from a business operation point of view. With fine-grained fault tolerance, bubble execution can recover from failures while maintaining deterministic behavior.

In summary, the main contributions of this paper are:

- Bubble execution, a new query processing framework for interactive workloads at cloud scale. The framework consists of key extensions to the cost-based query optimizer and the job scheduler components.
- A cost-based optimizer policy for bubble generation, with horizontal and vertical cut heuristics to balance latency and failover cost.
- A two-phase scheduling algorithm to avoid deadlocks and under-utilization of resources.
- A fine-grained fault tolerance mechanism that leverages direct operator-to-operator pipe channels and recoverable channels.
- A comprehensive performance evaluation on a large-scale production system showing the benefits of bubble execution over batch and gang execution.

The rest of this paper is structured as follows. Section 2 presents an overview of the framework and architecture for bubble execution. Section 3 provides our new proof that bubble generation is an NP-Hard problem, and presents details of a bubble generation algorithm necessary to achieve low query latency within resource constraints. Section 4 presents a scheduling mechanism to achieve scheduling efficiency with high resource utilization. Section 5 describes how bubble execution achieves fault tolerance with minimal performance impact. Section 6 presents performance experiments comparing bubble execution to other execution models, provides evidence for the efficiency of its fault tolerance strategy, and evaluates the system scalability. Section 7 reviews related work, and Section 8 provides a summary and conclusions of this work.

2. BUBBLE EXECUTION FRAMEWORK

Bubble execution is built on the top of JetScope architecture, which scales out interactive query processing over big data while supporting hundreds of concurrent queries with a variety of complexities. Bubble execution extends two components of the architecture: the query optimizer to enable bubble generation, and the job manager to enable smooth scheduling of bubbles without wasting resources and avoiding deadlocks.

2.1 Architecture

In this section, we describe the overall architecture with bubble execution and how a query is answered, from submission, compilation, optimization to execution and returning the results in a streaming fashion.

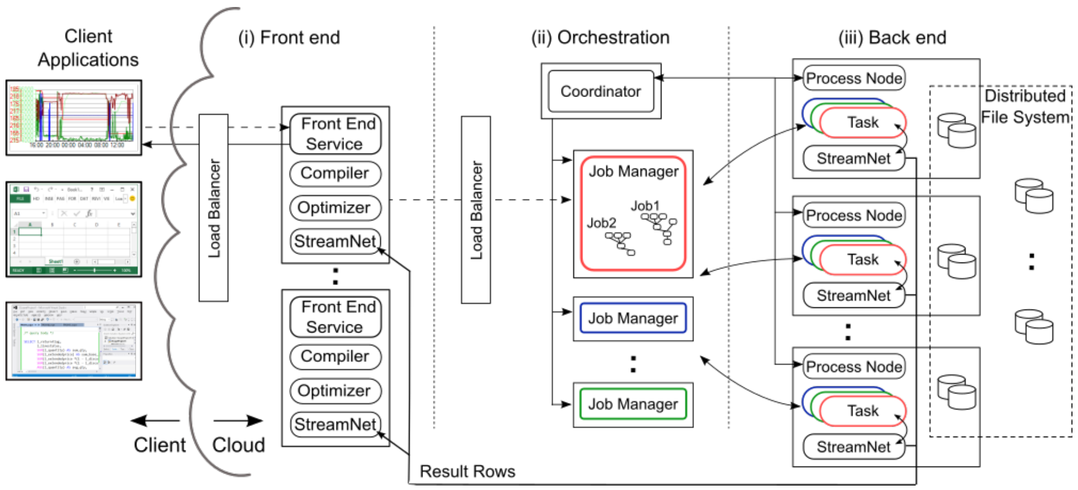


Figure 1: JetScope Architecture Overview

In JetScope, queries are submitted to the cluster portal either from users' development environments or various applications via ODBC APIs. Inside the computing cluster, the system is comprised of three major layers: (i) front-end, (ii) orchestration, and (iii) back-end, as shown in Figure 1. 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 nodes for execution. The data is read from a distributed file system, storing data across the cluster. Whenever possible, tasks are dispatched to the nodes 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 become available. To support high query concurrency, the system automatically shards query compilation and scheduling among different instances of system components and provides efficient load balancing among them. The capacity of the system can be dynamically adjusted by adding or removing compute nodes in various functions.

Bubble execution improves the layers (i) and (ii) of the above architecture, specially involving the following components.

- A *compiler* that takes scripts as input, and generates a distributed computing plan. The script is written in Scope, a SQL-based language, to provide a more user-friendly interface for the cloud developers and data scientists to process and analyze TBs of data at great flexibility.
- A cost-based *query optimizer* that partitions a computing plan into small tasks. In addition, the optimizer does task re-ordering, redundant task elimination, and extracting task prediction. The optimizer builds a cost model, searches the solution space and aims to get the best distributed execution plan in bubble groups.
- A *job manager* that dispatches the tasks in the plan to selected compute nodes, collects the execution state from the nodes, and takes different strategies to handle the cluster dynamics like duplicate task execution and re-invocation.

2.2 Token

We use the concept of a token to control resources usage in query execution. In Scope, a query is compiled and cost-based optimized into a query execution plan (QEP). The QEP is a directed acyclic graph (DAG), where data is processed from the leaf vertices representing file and structured stream scans, through various relational operators (e.g., filters, joins, aggregations), toward a final resultset. Each vertex (relational operator) in the DAG is called a task. Tasks are organized into stages, where each stage has one or more tasks that execute the same operation over different files, table partitions, or intermediate results.

In Cosmos, a physical server resource, is divided into a set of logical resource units, each called a *token*. For example, a server with 24 CPU cores and 128 GB RAM, can reserve 8 cores and 32 GB RAM for platform services, and leave 16 cores and 96 GB RAM for user query workloads. In this example, a token represents a resource slice of 1 core and 6 GB RAM. A goal of the query optimizer is to map one token per query operator (task) in the QEP, which means, a physical machine can process 16 QEP tasks concurrently. A user submits a query to the system giving it a token budget, based on the importance or priority of the query to the users business goals. The token budget for a query is fixed throughout the execution of the query. The job manager takes the QEP and token budget for a query, and orchestrates the execution of the QEP by stages, each of which consumes the assigned token budget per query.

2.3 Bubble and Channel

A bubble consists of one or more tasks that can be executed all at once. Connected tasks within a bubble have a strong data communication dependency, and in-memory streaming data between them, avoiding disks, reduces latency. A *channel* [18] is the abstraction of communication between two tasks. One task as producer can write data into the channel, and other tasks as the consumers can read data from the channel. In the bubble execution framework, there are two types of channels:

- **Pipe channel**: provides one-time streaming capability between tasks. The connected tasks can execute in streamline manner. But it's not recoverable on any

Execution model	Scheduling granularity	Execution granularity	Channel	FT requirement for channels
Batch	Task	Group of tasks	Recoverable	All channels recoverable
Bubble	Bubble	Group of bubbles	Recoverable and Pipe	Channels between bubbles recoverable
Gang	All tasks	All tasks	Pipe	All channel recoverable or none (restart all)

Table 1: Execution model comparison

failure. In JetScope, the pipe channel is implemented as a service, called StreamNet, with memory cache that can avoid landing data to disk.

- **Recoverable channel**: persists the data from the producer and ensures that the consumer can read the data from any node at any time, even when a consumer restarts. This kind of channel is the basis for failure-tolerance. Failed tasks can be re-executed once detected. The most common implementation of the recoverable channel is a disk file. For better reliability, it could be a stream on a distributed file system with multiple replicas.

Inside a bubble, tasks are connected via pipe channels, so that once in execution they can in-memory stream data between them. Between the bubbles, tasks are connected via recoverable channels, which provide a checkpoint for the bubble to recover if any failure occurs. A query consists of one or more bubbles which can execute concurrently depending on the dependency graph and availability of compute resources.

The Microsoft Cosmos service supports a batch data preparation job service that uses a batch execution model. We can consider batch execution as one end in the spectrum of bubble task density, where a query contains as many bubbles as tasks, that is, there is one bubble per task. Each task is connected via a recoverable channel. JetScope, an interactive service, introduced gang execution which represents another end in the bubble task density spectrum consisting of a single execution bubble for all the tasks comprising the query. Bubble execution represents an intermediate point in the task density spectrum. Table 1 summarizes those three execution models.

2.4 Extensions of Optimizer and Job Manager

With the new challenges on cost model and scheduling policy from the bubble and channel, bubble execution extends two components of the JetScope architecture: the query optimizer to enable bubble generation, and the job manager to enable smooth scheduling of bubbles without wasting resources and avoiding deadlocks.

The optimizer performs its usual cost-based optimization [5] producing a distributed query execution plan represented by a DAG. We propose adding a bubble-generation phase which breaks down the DAG into groups of tasks, called bubbles. The input to this phase is the QEP DAG, the per-query token budget, and the estimated data size carried by each channel connecting tasks. There are many approaches for traversing the QEP DAG and defining the bubble boundaries. In our implementation, we propose a heuristic, described in details in Section 3.2, based on iterative vertical and horizontal cuts of the graph until we reach bubble sizes that meet the query token quota and minimize latency. We have found our heuristic strikes a practical engineering balance that adds little overhead to the optimization time while producing results that are comparable to gang scheduling.

The job manager takes the output of the query optimizer which consists of a QEP DAG annotated with bubble boundaries. Each bubble must consume no more than the total number of tokens given to the query. However, bubbles could potentially require a fraction of the total allocated token budget which opens the possibility of scheduling multiple bubbles of a query concurrently subject to the data dependencies implied by the QEP DAG. The responsibility of job manager is to find a schedule of bubbles that honors data dependencies between tasks minimizes waste of computing resources. Achieving efficient bubble scheduling in a distributed environment is challenging since all concurrent queries at a given point in the system are competing for the same physical resources. In JetScope, which uses gang execution, running multiple queries concurrently without resource control can result in deadlocks [3]. In bubble execution, the job manager is extended with a two-phase scheduling algorithm that avoids such deadlocks. This algorithm is described in Section 4.

2.5 Resource Management

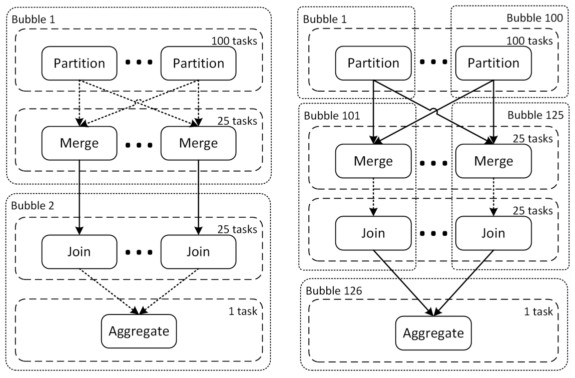
When the optimizer partitions the plan into small tasks, all of them have a similar workload and each of them fits a token of resources. A task in execution maps to a single token for the duration of the task. Based on the amount of resources given to the query, a certain number of tokens are allocated to execute it. The job manager uses these tokens to schedule tasks on the cluster resources. At any given moment in execution, job manager strives to keep a maximum number of concurrently running tasks as similar to the number of tokens allocated to the query. However, depending on the stage of the query processing, there may not be enough tasks to fully utilize the tokens, similarly, there can be more tasks than that can be executed at once.

For batch execution, a query can use one or more tokens for execution. The job manager computes the task priorities according to task dependencies, and dispatches the tasks to the matched nodes under the given tokens. To further reduce the query latency, gang execution dispatches all the tasks at once for an execution plan, enabling related tasks to stream data between them. However, gang execution requires that all tokens for the entire execution plan are available to the job manager. Further, it keeps all tasks of the query in execution even if the consumer task may be idle and waiting for the data to be available from its producer tasks. This aggressive resource allocation will prevent other queries from executing due to lack of tokens in the whole system. Bubble execution strikes a balance between the batch and gang execution approaches, making the much needed parts of the query plan to use streaming while adhering to token (resource) constraints.

```
NoComplainOrders =
    SELECT *
    FROM orders
    WHERE NOT (O_COMMENT.Contains("special")
    AND O_COMMENT.Contains("packages"));

SELECT c_count,
       COUNT(*) AS custdist
FROM
(
    SELECT C_CUSTKEY, COUNT(O_ORDERKEY) AS c_count
    FROM customer
    LEFT OUTER JOIN NoComplainOrders
    ON C_CUSTKEY == O_CUSTKEY
    GROUP BY C_CUSTKEY
)
GROUP BY c_count
ORDER BY custdist DESC, c_count DESC;
```

Figure 2: TPC-H Q13 script



(a) 125 tokens (b) 10 tokens

Figure 3: TPC-H Q13 execution plans

2.6 A Sample Query and the Bubbles

Figure 2 shows the script of TPC-H Query 13 and figure 3 shows its possible plans. During the compilation, the optimizer partitions the plan into 151 tasks: 100 tasks to partition the table orders, 25 tasks to merge the partitioned result, 25 tasks to join with the table `customer`, and one task to aggregate the final result, as depicted in Figure 3a.

Given 125 tokens, the optimizer can group the tasks into two bubbles: The first bubble contains 100 `Partition` tasks and 25 `Merge` tasks; and the second one includes the remaining tasks. During the plan execution, the scheduler first dispatches 125 tasks with 125 tokens to achieve the maximally allowed parallelism. After all the 125 tasks complete, it dispatches the rest 26 tasks all together. If the constraint is 10 tokens, the plan could be broken down into small 126 bubbles, as shown in Figure 3b.

3. BUBBLE GENERATION

Bubble generation is the process of breaking down an execution plan into bubbles. An execution plan is composed of tasks and the tasks are organized as *stages* [18]. Every stage has one or more tasks which execute the same operations against different data partitions. A bubble can span over several stages, and the tasks inside a bubble can have different operations. Bubble generation can be abstracted as an optimization problem with given token budget and channel weights. The section 3.1 first formalizes this problem and proves its NP hardness. And then the section 3.2 proposes a heuristic greed algorithm to provide an approximate solution.

3.1 The Minimum DAG K-cut Problem

We capture the essential optimization required for generating the bubbles as the MINIMUM DAG K-CUT PROBLEM. This problem considers one iteration, where you are required to find just one bubble, of a given size $k <= K$, the token budget. The problem asks for a bubble of size k, with the goal of minimizing the data persistence cost. We state the problem as a decision problem, since our main result here is to show NP-Hardness of this problem.

DEFINITION 1 (THE MINIMUM DAG k-CUT PROBLEM). *Given a directed acyclic graph (DAG) $G = (V, E)$, and integers k and C, is there a set of vertices $S \subseteq V$ such that*

1. $|S| = k$,
2. *there are no edges directed from S^c to S, (where $S^c = V \setminus S$,) and*
3. $|E(S, S^c)| \leq C$, *where $E(S, S^c)$ is the set of edges (u, v) such that $u \in S$ and $v \in S^c$?*

The problem can be generalized to a weighted version, where each edge $e \in E$ has a capacity c_e and we require an upper bound on the capacity of the cut, i.e.,

$$c(S, S^c) = \sum_{e \in E(S,S^c)} c_e \leq C.$$

Note that this is different than the classic min k-cut problem [14] because of two reasons:

1. The min k-cut problem requires a cut that results in k connected components, whereas the k in our definition refers to the size of the cut.
2. There is an additional requirement in our problem that the cut is a *topological* cut in the graph.

We will show that this problem is NP-Hard via a reduction from the CLIQUE problem.

DEFINITION 2 (THE CLIQUE PROBLEM). *Given an undirected graph $G = (V, E)$ and an integer l, is there a clique in G of size l, i.e., is there a set of vertices $S \subseteq V$ such that*

1. $|S| = l$, *and*
2. *every pair of vertices in S have an edge between them?*

It is well known that CLIQUE is NP-Hard, and was one of the original 21 problems shown to be NP-Hard by [19].

We observe that a standard reduction allows us to assume that the input graph for CLIQUE is *regular*, i.e., all vertices have the same number of edges; see [9]. We will use this version for the reduction.

THEOREM 1. *The DAG k-cut problem is NP-Hard*

PROOF. Suppose that we are given an instance of regular-CLIQUE, which is a d-regular graph $G = (V, E)$, i.e., each vertex has exactly d edges incident on it, and an integer l. The problem is to decide whether there is a clique of size l in G.

We construct an instance of Min DAG k-cut as follows. This instance has a DAG $G' = (V', E')$ where $V' = V \cup E$. There is one vertex in V' for every vertex as well as every edge in G. There is an edge $e' = (u', v') \in E'$ if and only if u' is a copy of a vertex $u \in V$, v' is a copy of an edge $e \in E$, and e is incident on u in G. The parameters k and C in the instance are set as $k = l + \binom{l}{2}$, and $C = (d-l+1)l = dl - 2\binom{l}{2}$.

The following claim completes the proof.

Claim: There is a clique of size l in G if and only if there is a DAG k-cut of size $\leq C$ in G'.

CLIQUE $\Rightarrow$ DAG k-cut Assume that there is a clique in G of size l. Let this clique be on the set of vertices $S \subseteq V$. Consider the following cut, given by the set $S' \subseteq V'$ that includes all copies of vertices and edges in the clique S. Clearly the size of S' is k. Consider the edges that leave S'. These are in 1:1 correspondence with edges in E such that one of their endpoints is in S and the other isn't. Each vertex in S has exactly d edges in all, out of which exactly $l-1$ have another endpoint in S itself (since S is a clique). That leaves us with $d - l + 1$ edges that leave S for each vertex in S, giving us exactly C edges in all.

DAG k-cut $\Rightarrow$ CLIQUE Consider any k-cut in G', say given by $S' \subseteq V'$. We will argue that if S' doesn't look like the construction in the previous paragraph starting from a clique of size l in G, then the number of edges going out of S' must be more than C. Suppose not.

First, there must be at least $l + 1$ vertices in S' that correspond to vertices in V. Now we just count the number of edges that cross the cut S'. Each vertex in S' that corresponds to a vertex in V has exactly d outgoing edges in G'. For each vertex in S' that corresponds to an edge in E, there are exactly 2 incoming edges in G'. Since there are at most $\binom{l}{2} - 1$ such vertices in S', they can together save at most $2\binom{l}{2} - 2$ such edges. Thus the number of outgoing edges from S' is at least $d(l+1) - (2\binom{l}{2} - 2) = dl - 2\binom{l}{2} + d + 2 > C$. $\square$

3.2 A Heuristic Algorithm

Although bubble generation is different from min k-cut problem [27] since the k is the token budget, the classic algorithms for min k-cut problem can still be applied for bubble generation with some modification. For example, we implemented the algorithm SPLIT [25] and found this classic greedy algorithm tends to generate superfluous bubbles and compromise the query execution latency. Based on the heuristics from our engineering practices, we propose a greedy bubble generation algorithm, called MERGE. This algorithm leverages two kinds of cuts to generate the bubbles:

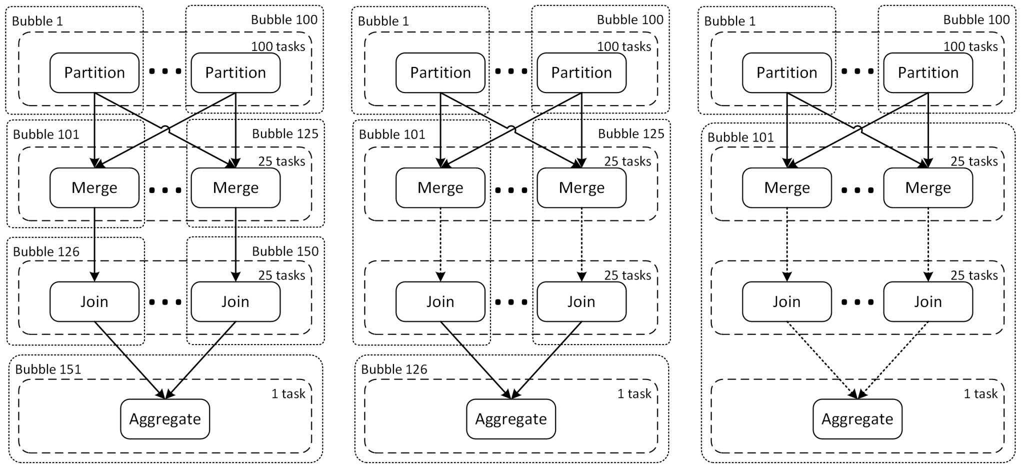
- **Horizontal cut**: made between two dependent stages. Like Figure 3a, a four-staged plan is cut between `Merge` and `Join` stages, into two bubbles. A horizontal cut determines where to place the recoverable channels.
- **Vertical cut**: made between those tasks with no direct or indirect dependencies within stages. For instances, in the Figure 3b, besides two horizontal cuts (one between `Partition` and `Merge` stages and the other between `Join` and `Aggregate` stages), there are also two vertical cuts: the first is made within the `Partition` stage so that all tasks are partitioned into 100 bubbles; the other cut breaks the `Merge` and `Join` stages into 25 independent bubbles. Each of the bubbles (101∼125) has one task from the `Merge` stage and one from the `Join` stage.

```
Input: Token Budget, Tasks and Channels with
       estimated intermediate data size
Output: Bubbles
/* init                                              */
Assign each task to a bubble;
/* Merge while keeping vertical cut                  */
repeat
    Sort channels by intermediate data size
      descendingly;
    foreach channel do
        if (vertical cut constraint) and (token budget
          constraint) and (heuristic constraint) then
            Merge the producer and consumer bubbles;
            break;
        end
    end
until No bubbles can be merged further;
/* Merge to break vertical cut                       */
repeat
    Sort channels by intermediate data size
      descendingly;
    foreach channel do
        if (token budget constraint) and (heuristic
          constraint) then
            Merge the producer and consumer bubbles;
            break;
        end
    end
until No bubbles can be merged further;
```

Algorithm 1: Algorithm MERGE for bubble generation

Algorithm MERGE 1 starts with a basic solution: every task is a bubble; then it enumerates each channel, and merges the connected bubbles while keeping the vertical cuts; finally, it enumerates each channel again, and tries to merge bubbles as many as possible. The intermediate data size of two connected bubbles determines the priority of merging. And each bubble merging must honor the constraint of token budget. Additional heuristic is also incorporated as the constraint of bubble merging. For example, a `partition` stage without filter operation often generates large intermediate data that stress pipe channels. So if the token budget is small, a horizontal cut between the `partition` stage and its downstream stage is preferred and the two bubbles containing the two stages cannot be merged.

Figure 4 shows an example of bubble generation for TPC-H Query 13 under 100 token budget limit. First, this algorithm makes a vertical cut on each stage so that each task is a bubble. Next, it searches for any two stages that have a one-to-one dependency and merges the bubbles within stages so that the vertical cut still holds. This is to merge bubbles in the vertical dimension. At last, the algorithm merges as many bubbles as possible while keeping the bubble size within token budget. In other words, it merges bubbles



(a) Take each task as a bubble. (b) Merge bubbles vertically. (c) Merge bubbles horizontally.

Figure 4: TPC-H Q13: bubble generation steps with 100 tokens

in the horizontal dimension. This greedy algorithm doesn't generate the best bubbles, but it's good enough for current production clusters. Section 6 provides the evaluation result for the algorithm MERGE.

4. BUBBLE SCHEDULING

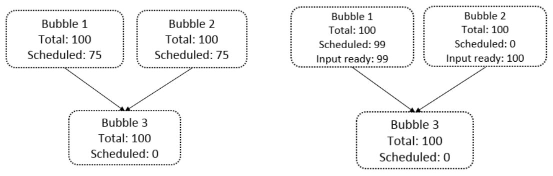
The stage topology of an execution plan contains a skeleton that represents how tasks are connected to each other. Bubble generation draws the bubble boundaries by annotating the channels between stages and the job manager follows the annotations to materialize the execution plan and identifies bubbles. It then dispatches those bubbles to computing nodes in an efficient way.

4.1 Two-Phase Scheduling

Achieving efficient bubble scheduling in a distributed environment is challenging. In gang execution, running multiple queries concurrently without resource control could result in deadlocks [3]. To solve this, gang execution introduces query admission control, which accepts a query only when resources are available to execute the whole query at once.

A task is ready only if all of its inputs are ready to be consumed and a bubble is ready only if all the tasks in the bubble are ready. In batch execution, a ready task is enqueued to wait for resources and de-queued once resources are ready. In bubble execution, the amount of available resources is dynamic and at some moment may be less than the required amount from a ready bubble. To efficiently utilize available resources, the job manager must solve the following two challenges:

- **Resource competition from different bubbles**: consider the example shown in Figure 5a. There are 150 tokens available, and two bubbles waiting for dispatching, each requires 100 tokens. The first 75 tasks in Bubble 1 are ready and en-queued, then another 75 tasks in Bubble 2 are ready and en-queued. The 150 tasks could be dispatched, but there will be no resources to dispatch any other tasks in Bubble 1 or Bubble 2. Bubble 1 and Bubble 2 will wait for each other in a deadlock.



(a) Deadlock (b) Under-utilization

Figure 5: Scheduling challenges

- **Resources under-utilization**: Figure 5b show another example with total 150 tokens and two bubbles requiring 100 tokens for each. Assume 99 tasks in Bubble 1 are ready and one task is left because its upstream task is a straggler. Even when all tasks in Bubble 2 are ready, the queue can be locked by Bubble 1 to ensure it is dispatched prior to Bubble 2. In this case, the system is wasting resources due to the slow tasks.

To solve the resource competition and under-utilization issues, we propose two-phase scheduling algorithm that uses two types of queues: a per-bubble *bubble queue* and a *global queue*. Any task that becomes ready is en-queued into it's bubble queue first. Whenever a bubble queue is full with all the tasks in the corresponding bubble, it pops them out and pushes them into the global queue. In this way, no matter how many resources are available, the tasks in the global queue are always ordered by bubbles. Even when available resources are less than the required amount of a ready bubble, the job manager can still dispatch the tasks in global queue without waiting for more resources to become available. After all, the final dispatching order for all tasks is eventually consistent with the bubble order.

When a task fails in a given bubble, the failure will be propagated through the pipe channels to the other tasks in the same bubble. This will further cause all the tasks in that bubble to be re-run, and hence they will be first moved into the bubble queue in the order of the failure arrival time. Next, following the two-phase scheduling logic above, those tasks will be pushed to the global queue when the bubble

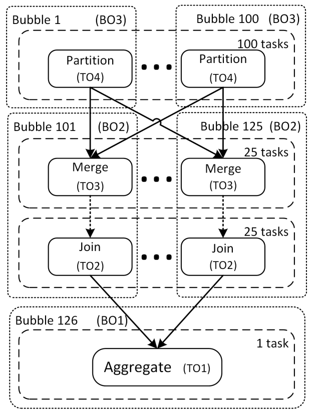


Figure 6: Query 13: priority assignment

queue is full. Thus, the task dispatching is always bubble atomic with the two queues.

4.2 Bubble Priority

In the two-phase scheduling algorithm, when a bubble is ready, the tasks in the bubble are all in the global queue. If there are multiple bubbles in ready state, dispatching tasks strictly in FIFO order does not yield the best resource utilization. In a complex query consisting of thousands of tasks, the tasks in the critical path usually determine the end-to-end latency of the whole query owing to topological dependencies. Especially, when some tasks fail and are moved back into the global queue, they should have higher priorities to dispatch so the query execution would not be blocked by the failures. To address this issue, the tasks are assigned with priorities. When multiple tasks are popped from the global queue, they are sorted by their priorities, and then dispatched in that order.

The priority of a task contains three factors:

1. **Bubble order**: is the topological order in the bubble dependency graph. The bubble with higher order should be dispatched earlier.

2. **Bubble ID**: is a unique integer for each bubble. During the sort, bubble IDs ensure the tasks are grouped by bubbles so the dispatching is bubble-atomic.

3. **Task order**: is the topological order in the task dependency graph. Even in the same bubble, tasks with higher order should be dispatched earlier to build the output channels.

When the scheduler sorts tasks, it compares tasks by the three numbers above in the listed order.

Figure 6 illustrates the priority assignment for TPC-H Query 13. Bubble 1∼100 has the order of 3; Bubble 101∼125 has 2; and Bubble 126 has order of 1. On the task level, the order for `Partition`, `Merge`, `Join` and `Aggregate` are 4, 3, 2, 1, respectively. The dispatching order can be determined if the global queue has some tasks like the following cases:

- `Partition` task in bubble 100, and `Merge` task in bubble 101: `Partition` task will be dispatched earlier.

- `Partition` task in bubble 1 and `Partition` task in bubble 100: Although the two tasks have the same bubble order, `Partition` task in bubble 100 is dispatched first due to a bigger ID number.

- `Merge` task and `Join` task in bubble 101 and `Merge` task and `Join` task in bubble 125: ordering by bubble level first and task level second, the dispatching sequence is `Merge` task in bubble 125, `Join` task in bubble 125, `Merge` task in bubble 101, and `Join` task in bubble 101.

5. FAULT TOLERANCE

Failures and workload fluctuations are the norm in 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. The most common failover mechanism is to rerun a task after it fails. To be able to rerun a task, there are two requirements to be met:

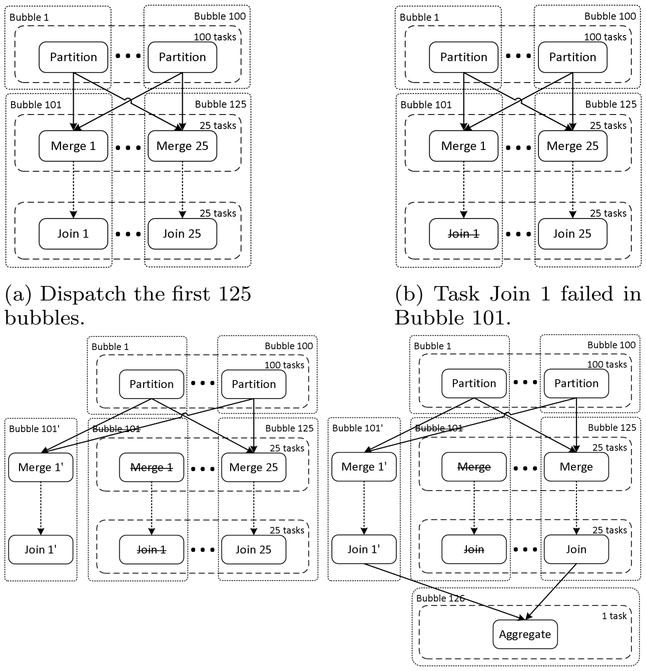
- **Channel recover-ability**: every channel can be read at any point. This implies data persistence. So the data can be read and sought. A pipe channel doesn't provide such a capability.

- **Deterministic behavior**: intermediate results written to channels are deterministic. This means any task running twice should generate the same outputs. In the real world cloud-scale computing system, some operations are designed to be random to maximize performance. Also, it's hard to avoid non-deterministic behavior from user defined operations.

If one task fails, the re-run task reads inputs from the recoverable channels, output channels are rebuilt, and the downstream consumer continues reading the data by skipping the duplicated part through the seek operation.

Compared to bubble execution, with gang execution, it is barely possible to resolve non-determinism issues in an elegant way during the failover process. Since the consumer task has been reading partial data from the failed task for a while, the only way to ensure computing correctness is to build up record sync mechanisms as used in streaming systems e.g., Kafka [12], Storm [13], StreamScope [23], so that the re-run task can inform its consumers of the need to discard dirty records. However, it is inappropriate to add this overhead to a non-streaming scenario.

Bubble execution simplifies the complexity of the failover mechanism by addressing the challenge in a coarse granularity. All tasks in a bubble are dispatched at once. Recoverable channels are built between bubble boundaries and pipe channels are adapted inside bubbles. So the bubble boundary is actually the checkpoint for the recovery. The advantages of this straightforward design are: (a) A pipe channel is usually implemented as a FIFO buffer based on memory. Without the requirement of persistence, it's easy to achieve high performance and low resource utilization. (b) Non-determinism is not a concern because the bubble is recovered as a unit to avoid partial data reads. Considering the impact of non-deterministic tasks, the optimizer can even place those tasks at the bubble boundary to cause outputting to enhanced recovery channels, like persisted streams with three replicas. This optimization reduces bubble re-run costs in case of a channel failure. (c) The job manager has less communication with the computing nodes so that it can have more resources available to



(a) Dispatch the first 125 bubbles.

(b) Task Join 1 failed in Bubble 101.

(c) Bubble 101′ is created to substitute Bubble 101.

(d) Dispatch Bubble 126 after all its producer bubbles finishes.

Figure 7: Failover steps for bubble execution

support additional concurrent queries, compared to the fault handling mechanism in JetScope.

To lower this cost, our bubble generation algorithm always begins with a high number of small bubbles, and expands bubbles while maintaining vertical cuts as much as possible when growing bubble sizes slowly, as was shown in Section 3.2.

Figure 7 illustrates how the failover works for the plan from Figure 3b: (a) the first 125 bubbles are dispatched for execution; (b) task `Join 1` failed in bubble 101; (c) bubble 101′ is created for re-execution and the input are read via recoverable channels; (d) bubble 126 is dispatched when all its dependent bubbles complete.

6. EVALUATION

We performed detailed experiments on bubble execution to evaluate its performance and resource usage under various scenarios. In this section, we start with experimental results in a real production system, using 10TB TPC-H queries to drill down into query latency, plan quality, scheduling efficiency, fault tolerance and scalability. Experiments where run on a production subcluster consisting of 200 compute nodes with a total capacity of 2000 tokens. Each compute node consists of a server with 16 cores, 128 GB RAM, 200MB/s Disk IO, and a 10 Gbps NIC.

6.1 Query in Production

The system has been deployed to hundreds of servers in production at Microsoft, serving millions of queries against a wide range of big datasets daily. Most of the queries are simple ones which require less than 10 tokens. We did observe large queries requiring more than 10,000 tokens, as shown in Figure 8, which illustrates the token distribution among

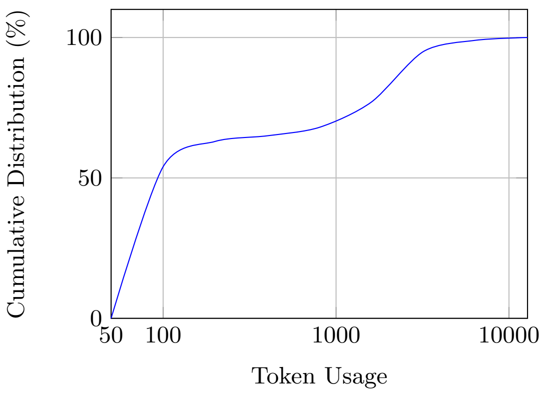


Figure 8: Cumulative token distribution from production cluster

queries that require more than 50 tokens. We expect that the number of large queries will increase with the adoption of bubble execution.

6.2 Latency and Resource Utilization

We set up the experiment to run Query 5 with bubble execution and batch execution under different token budgets, as shown in Figure 10. For the same token budget, bubble execution gains shorter latency compared to batch execution. As the token budget increases, bubble execution reduces latency from 130 to 75 seconds. Bubble execution with 500 tokens shows a slowdown compared to adjacent plans. This is because our greedy bubble generation algorithm produces a plan with a smaller max task degree of parallelism (DOP). The batch execution gains the same performance with tokens from 200 to 1300 because the max task DOP is 200. Gang execution requires 1316 tokens and generates the best performance in latency.

We did a set of similar experiments on all TPC-H queries. Each query was run under different tokens varying from 10% to 90% of the token usage for gang execution. The average speedup is illustrated in Figure 11. If we use 10% of resources, we slow down the queries by 3x (34% of speedup). The speedup becomes higher as the resource usage increases. With 50% of usage, the slowdown is less than 25%. This gives system administrators configuration flexibility to trade-off between latency and request concurrency, from a business operation point of view.

We also collected the numbers for batch execution under the half resources as shown in Figure 9. Most queries achieve similar performance with bubble execution given the half resources used by gang execution. For Q3, Q9, Q15 and Q19, performance is almost on par with gang because the intermediate data between bubble boundaries is very small. All the tasks in Q14 generate few data so even batch execution can provide the same performance. The queries Q10, Q16, Q17, Q20 and Q22 are examples where the vertical-cut-first heuristic builds up a bubble boundary on the channels with huge intermediate data, so there is limited speedup from streamline execution. The fan-out queries, Q1, Q4, Q6 and Q12, are exceptions because the bubble generation can only produce single-task bubbles to meet the resource limitation. Thus, bubble and batch execution show similar performance, and streamline execution brings huge potential to boost the

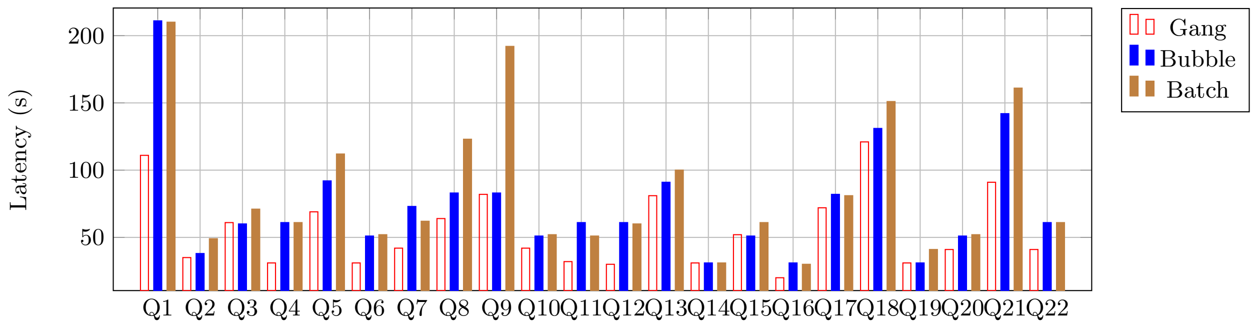


Figure 9: Performance comparison among different execution models

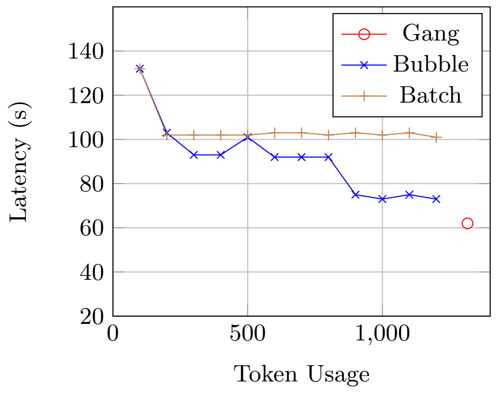


Figure 10: Query 5 running with different token budgets

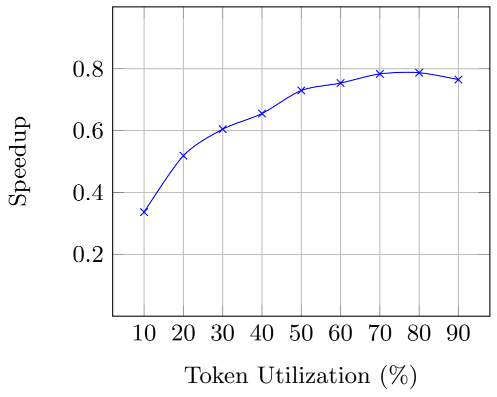


Figure 11: Average speedup v.s. resources for TPC-H queries

queries.

6.3 Plan Quality

Since bubble generation is an NP-hard problem, the algorithm MERGE tries to provide an approximate solution with the heuristics from engineering practices. To validate its efficiency, we implemented a modified version of the classic greedy algorithm for the min k-cut problem as the baseline. This baseline algorithm recursively applies algorithm SPLIT [25] on the sub graphs that don't fit the token budget until all the bubbles can run within the token budget. We compared the latency of query execution for the plans from our bubble generation algorithm against the baseline algorithm.

Figure 12 shows the speedup of the plan generated by the algorithm MERGE over the baseline algorithm. Among 22 TPC-H queries, our algorithm show average 1.17x speedup on the baseline; Q1, Q4, Q6 and Q12 have the same plan from the two algorithms, and thus the same latency; Q8, Q18, Q20 and Q21 show average 1.10x slowdown. Algorithm MERGE is more efficient because it tends to utilize the full token budget for the computation and thus generates less bubbles; while the algorithm SPLIT always naively takes the minimum cut and results in more bubbles. Figure 13 illustrates the different plans by the two algorithms for Q13. The baseline algorithm generates 126 bubbles while the algorithm MERGE generates 101 bubbles and manifests better performance on streamline execution.

6.4 Scheduling Efficiency

Token usage	Average DR	Average WR
250	99.5%	9.3%
288	99.9%	10.2%
300	98.5%	14.1%
350	98.1%	14.8%
400	99.7%	6.9%

Table 2: Schedule efficiency metrics

Although an execution plan can be decomposed into many tasks, not all of them can run in parallel. Based on the dependency graph on task level, the amount of required resources varies during the query execution. Figure 14 shows an example. Given an execution plan, the required tokens increase gradually from the beginning and decline until the query completes. In gang execution, the token usage could be very high at the beginning. The bubble execution always tries to use all available tokens up to the provided limitation; while batch execution cannot due to smaller dispatching unit and task dependencies.

We evaluate the scheduling efficiency of bubble execution by two rates: (i) **dispatching rate** measures how efficiently the available resources are utilized by bubbles; (ii) **waiting rate** indicates the waiting time for the available resources to be used by a task.

Dispatching rate is defined as:

$$DR_t = \begin{cases} 1 & \text{if } GQ_t = 0 \\ \frac{min(AT_t, GQ_t)}{TT} & \text{if } GQ_t > 0 \end{cases},$$

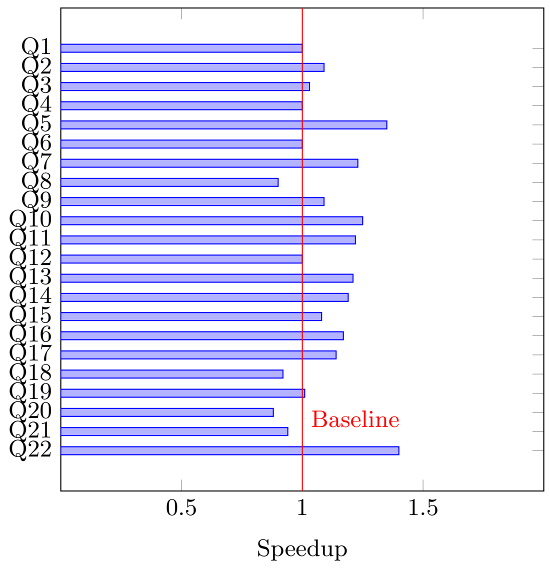


Figure 12: The efficiency of algorithm MERGE

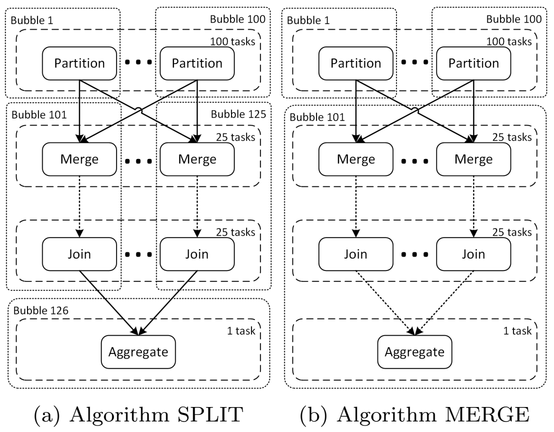


Figure 13: TPC-H Q13: plans with 75 tokens

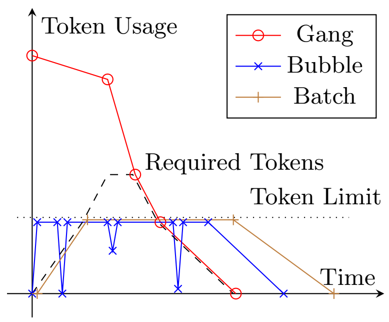


Figure 14: Resource utilization during query execution

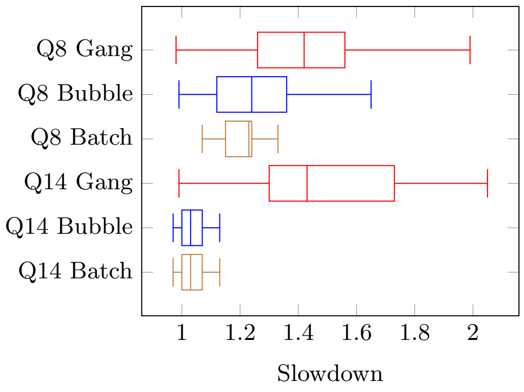


Figure 15: Execution slowdown due to failover

where AT_t is the number of available tokens, TT is the total amount of the given tokens, and GQ_t is the number of tasks in the global queue. AT_t and GQ_t are two dynamic numbers varying along with query execution.

Waiting rate is defined as:

$$WR_t = \begin{cases} 0 & \text{if } BQ_t = 0 \\ \frac{min(AT_t, BQ_t)}{TT} & \text{if } BQ_t > 0 \end{cases},$$

where BQ_t is the number of tasks in the bubble queues. WR_t is especially important for bubble execution. Thanks to two-phase scheduling, the ready tasks are en-queued into bubble queues, and then moved out. The waiting time for the queue to become full is non-trivial. In Figure 14, the token usage drops in a very short period for three times due to this. Batch execution is better here because it takes little time for the single-task bubbles to be ready.

We ran full TPC-H queries with different token budgets, and Table 2 shows the average number of these metrics. Regardless of the token usage, a 99% dispatching rate proves that the job manager is efficient at scheduling the tasks in the global queue. Compared to batch execution, there are 10% of resources idle when waiting for the rest of a bubble to be ready.

6.5 Failure Recovery

Fault tolerance is one important aspect for bubble execution. The penalty from one task failure consists of three factors: (1) Failure detection: some failures, like service crashing or local power outages, can be detected in sub-seconds; while other failures, like network congestion or machine overloads, take time to confirm. (2) Resource reallocation: in a multi-tenant system, each query has been assigned a portion of resources. Some failures, such as core service crashing, can cause a query to lose the control of its resources temporally. However, a query must recover based on its own resources to avoid using other tenants' resources. If there are no available resources, the query must wait for its capacity to recover. (3) Computing waste: it's inevitable to waste computing power, and it takes time to re-run the failed tasks.

For bubble execution, the last two factors are particularly related to the bubble generation. We assumed the failure detection time was zero, and injected one task failure for a query to evaluate the latency penalty for failover. Figure 15 shows the failover impact on two typical plans: Q8 and Q14. Bubbles for Q8 favor streamline execution, so the bubble size is relatively bigger. The latency penalty has a bigger range comparing to batch execution. Q14, on the other hand, has smaller bubbles which makes the bubble execution more like batch execution. So, the latency varies in a small range as for batch execution. Gang execution has the worst slowdown because any failure of a single vertex causes the whole query to rerun.

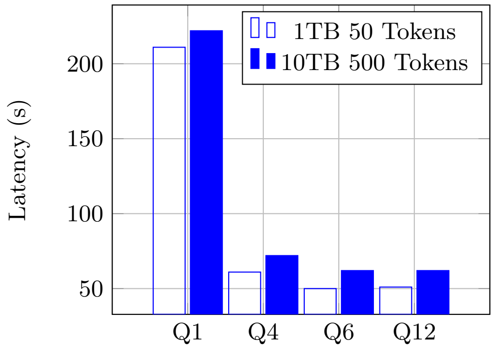


Figure 16: Bubble execution scalability

6.6 Scalability

To validate the scalability of bubble execution, we ran selected TPC-H queries (Q1, Q4, Q6 and Q12) with different scale factors and resources. The selected queries are all fan-out aggregations with two stages: an extract stage with several tasks reading data and pre-processing, followed by an aggregation stage with one task collecting all the outputs from the extract tasks and writing the final result. No matter what the scale factor is, the performance of those queries is linearly proportional to the number of tokens by nature. We prepared 1TB and 10TB data, and executed the queries with 50 and 500 tokens, respectively. Bubble execution reveals linear scalability as shown in Figure 16.

7. RELATED WORK

One challenge on batch processing at cloud scale is to ensure that a long-running job survives workload fluctuations and failures. Map Reduce [8], and Dryad [18] materialize intermediate data as recovery points resulting in jobs that can progress in spite of task failure. However, this approach trades execution performance for reliability. Bubble execution accelerates query execution by replacing some of the recovering points by pipe channels based on cost estimation.

Emerging interactive query engines focus on the latency of query execution to provide prompt response for user interaction. Map Reduce Online [7] was proposed to improve the performance by streaming data, which reduces the materialization cost. Dremel [24] provides a quick query execution engine based on columnar data. Shark [28] and Spark [29, 1] are built on resilient distributed dataset. It introduces lineage to track execution and recover from failure. JetScope [3] is optimized to reduce the processing latency by streamlining execution, yet it requires massive computing resources. Our work is based on JetScope and proposes bubble execution to enable streamline execution under limited resources.

Most of the modern distributed computing systems are inspired by transactional database systems which rely on a cost-based optimizer [16, 15] to generate efficient execution plans. Impala [20] implemented several plan optimizations, such as ordering and coalescing analytic window functions, and join reordering based on cost estimation. Scope [6, 30] provides a continuous query optimizer [5] which introduces a positive feedback from job execution to improve data statistics. Bubble execution introduces resource constraints as a new dimension to the cost model, and proposes a horizontal and vertical cut heuristic for bubble generation.

Resource control and capacity management are necessities in a multi-tenant system. Hadoop YARN [26] and Mesos [17] were developed to abstract resource management for different execution frameworks in a centralized manner. Apollo [4] introduces a cost-based scheduler which performs scheduling decisions in a distributed manner, utilizing global cluster information via a loosely coordinated mechanism. All those schedulers treat each task as a dispatching unit. Bubble execution extends the dispatching unit into a group, proposing a complete mechanism to deal with bubble management and scheduling fairness in a multi-tenant environment.

8. CONCLUSIONS

This paper introduced Bubble Execution, a novel query optimization and scheduling technique built on the existing JetScope architecture at Microsoft. It breaks a query execution graph into a collection of bubbles so that they can be scheduled for execution within per-query resource constraints. Moreover, it facilitates the use of in-memory streamed data transfers inside a bubble reducing the overall latency of query execution. Fault tolerance is enforced at bubble boundaries. These optimizations enable bubble execution to minimize latency on analytic queries while adhering to resource constraints. We believe that these concepts can be applied to any other cloud-scale data analytic engine of similar capabilities. The experimental data presented in the paper, confirms the benefits of bubble execution and highlight the trade-offs we made between latency, reliability, and resource constraints. Compared to gang execution, bubble execution could reduce resource usage dramatically while maintaining comparable query latencies.

9. ACKNOWLEDGMENTS

We would like to thank all the members of the Microsoft Big Data Analytics team for their contributions to the bubble execution framework and the Scope eco-system. We also acknowledge the contributions of the former members of the JetScope team including: Tao Guan, Xiaoyu Chen, Yongchul Kwon and Zhou Long.

At last, we would like to give our special thanks to José who was so passionate about bubble execution framework and gathered us for the paper. We are so sorry that you are no longer with us. Thank you for everything and rest in peace.

10. REFERENCES

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[2] P. A. Boncz, S. Manegold, M. L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[3] E. Boutin, P. Brett, X. Chen, J. Ekanayake, T. Guan, A. Korsun, Z. Yin, N. Zhang, and J. Zhou. JetScope: Reliable and interactive analytics at cloud scale.

Proceedings of the VLDB Endowment, 8(12):1680–1691, 2015.

[4] 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 *OSDI*, volume 14, pages 285–300, 2014.

[5] N. Bruno, S. Jain, and J. Zhou. Continuous cloud-scale query optimization and processing. *Proceedings of the VLDB Endowment*, 6(11):961–972, 2013.

[6] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *Nsdi*, volume 10, page 20, 2010.

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

[9] E. Demaine. Algorithmic lower bounds: Fun with hardness proofs. `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-890-algorithmic-lower-bounds-fun-with-hardness-proofs-fall-2014/lecture-notes/MIT6_890F14_Lec13.pdf`, 2014.

[10] Facebook. Presto. `https://prestodb.io/`.

[11] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and distributed Computing*, 16(4):306–318, 1992.

[12] A. S. Foundation. Apache Kafka. `http://kafka.apache.org/`.

[13] A. S. Foundation. Apache Storm. `https://storm.apache.org/`.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[15] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[16] G. Graefe and K. Ward. Dynamic query evaluation plans. In *ACM SIGMOD Record*, volume 18, pages 358–366. Acm, 1989.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[19] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[20] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, volume 1, page 9, 2015.

[21] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. Sql server column store indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1177–1184. ACM, 2011.

[22] L. Lin, V. Lychagina, W. Liu, Y. Kwon, S. Mittal, and M. Wong. Tenzing a SQL implementation on the MapReduce framework. 2011.

[23] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. StreamScope: Continuous reliable distributed processing of big data streams. In *Proc. of NSDI*, pages 439–454, 2016.

[24] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[25] H. Saran and V. V. Vazirani. Finding k-cuts within twice the optimal. In *Proceedings of the 32nd Annual Symposium of Foundations of Computer Science*, pages 743–751. IEEE, 1991.

[26] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[27] Wikipedia. Minimum k-cut. `https://en.wikipedia.org/wiki/Minimum_k-cut`.

[28] 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*, pages 13–24. ACM, 2013.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[30] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel databases meet MapReduce. *The VLDB Journal - The International Journal on Very Large Data Bases*, 21(5):611–636, 2012.