POLARIS: The Distributed SQL Engine in Azure Synapse

Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan

Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei

Jose Blakeley, Girish Dasarathy, Sumeet Dash, Lazar Davidovic, Maja Damjanic, Slobodan Djunic, Nemanja Djurkic, Charles Feddersen, Cesar Galindo-Legaria, Alan Halverson, Milana Kovacevic, Nikola Kicovic, Goran Lukic, Djordje Maksimovic, Ana Manic, Nikola Markovic, Bosko Mihic, Ugljesa Milic, Marko Milojevic, Tapas Nayak, Milan Potocnik, Milos Radic, Bozidar Radivojevic, Srikumar Rangarajan, Milan Ruzic, Milan Simic, Marko Sosic, Igor Stanko, Maja Stikic, Sasa Stanojkov, Vukasin Stefanovic, Milos Sukovic, Aleksandar Tomic , Dragan Tomic, Steve Toscano, Djordje Trifunovic, Veljko Vasic, Tomer Verona, Aleksandar Vujic, Nikola Vujic, Marko Vukovic, Marko Zivanovic

Microsoft Corp

ABSTRACT

In this paper, we describe the Polaris distributed SQL query engine in Azure Synapse. It is the result of a multi-year project to re-architect the query processing framework in the SQL DW parallel data warehouse service, and addresses two main goals: (i) converge data warehousing and big data workloads, and (ii) separate compute and state for cloud-native execution.

From a customer perspective, these goals translate into many useful features, including the ability to resize live workloads, deliver predictable performance at scale, and to efficiently handle both relational and unstructured data. Achieving these goals required many innovations, including a novel "cell" data abstraction, and flexible, fine-grained, task monitoring and scheduling capable of handling partial query restarts and PB-scale execution. Most importantly, while we develop a completely new scale-out framework, it is fully compatible with T-SQL and leverages decades of investment in the SQL Server single-node runtime and query optimizer. The scalability of the system is highlighted by a 1PB scale run of all 22 TPC-H queries; to our knowledge, this is the first reported run with scale larger than 100TB.

PVLDB Reference Format:
Josep Aguilar-Saborit, Raghu Ramakrishnan et al.
VLDB Conferences. *PVLDB*, 13(12): 3204 – 3216, 2020.
DOI: https://doi.org/10.14778/3415478.3415545

1. INTRODUCTION

Relational data warehousing has long been the enterprise approach to data analytics, in conjunction with multi-dimensional business-intelligence (BI) tools such as Power BI and Tableau. The recent explosion in the number and diversity of data sources, together with the interest in machine learning, real-time analytics and other advanced capabilities, has made it necessary to extend traditional relational DBMS based warehouses. In contrast to the traditional approach of carefully curating data to conform to standard enterprise schemas and semantics, *data lakes* focus on rapidly ingesting data from many sources and give users flexible analytic tools to handle the resulting data heterogeneity and scale.

A common pattern is that data lakes are used for data preparation, and the results are then moved to a traditional warehouse for the phase of interactive analysis and reporting. While this pattern bridges the lake and warehouse paradigms and allows enterprises to benefit from their complementary strengths, we believe that the two approaches are converging, and that the full relational SQL tool chain (spanning data movement, catalogs, business analytics and reporting) must be supported directly over the diverse and large datasets stored in a lake; users will not want to migrate all their investments in existing tool chains.

In this paper, we present the Polaris interactive relational query engine, a key component for converging warehouses and lakes in Azure Synapse [1], with a cloud-native scale-out architecture that makes novel contributions in the following areas:

- *Cell data abstraction:* Polaris builds on the abstraction of a data "cell" to run efficiently on a diverse collection of data formats and storage systems. The full SQL tool chain can now be brought to bear over files in the lake with on-demand interactive performance at scale, eliminating the need to move files into a warehouse. This reduces costs, simplifies data governance, and reduces time to insight. Additionally, in conjunction with a re-designed storage manager (Fido [2]) it supports the full range of query and transactional performance needed for Tier 1 warehousing workloads.
- *Fine-grained scale-out:* The highly-available micro-service architecture is based on (1) a careful packaging of data and query processing into units called "tasks" that can be readily moved across compute nodes and re-started at the task level; (2) widely-partitioned data with a flexible distribution model; (3) a task-level "workflow-DAG" that is novel in spanning multiple queries, in contrast to [3, 4, 5, 6]; and (4) a framework for fine-grained monitoring and flexible scheduling of tasks.
- *Combining scale-up and scale-out:* Production-ready scale-up SQL systems offer excellent intra-partition parallelism and have been tuned for interactive queries with deep enhancements to query optimization and vectorized processing of columnar data partitions, careful control flow, and exploitation of tiered data caches. While Polaris has a new scale-out distributed query processing architecture inspired by big data query execution frameworks, it is unique in how it combines this with SQL Server's scale-up features at each node; we thus benefit from both scale-up and scale-out.
- *Flexible service model:* Polaris has a concept of a *session*, which supports a spectrum of consumption models, ranging from "serverless" ad-hoc queries to long-standing pools or clusters. Leveraging the Polaris session architecture, Azure Synapse is unique among cloud services in how it brings together serverless and reserved pools with online scaling. All data (e.g., files in the lake, as well as managed data in Fido [2]) are accessible from any session, and multiple sessions can

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415545

access all underlying data concurrently. Fido supports efficient transactional updates with data versioning.

1.1 Related Systems

The most closely related cloud services are AWS Redshift [7], Athena [8], Google Big Query [9, 10], and Snowflake [11]. Of course, on-premise data warehouses such as Exadata [12] and Teradata [13] and big data systems such as Hadoop [3, 4, 14, 15], Presto [16, 17] and Spark [5] target similar workloads (increasingly migrating to the cloud) and have architectural similarities.

- *Converging data lakes and warehouses.* Polaris represents data using a "cell" abstraction with two dimensions: *distributions* (data alignment) and *partitions* (data pruning). Each cell is self-contained with its own statistics, used for both global and local QO. This abstraction is the key building block enabling Polaris to abstract data stores. Big Query and Snowflake support a sort key (partitions) but not distribution alignment; we discuss this further in Section 4.
- *Service form factor.* On one hand, we have reserved-capacity services such as AWS Redshift, and on the other serverless offerings such as Athena and Big Query. Snowflake and Redshift Spectrum are somewhere in the middle, with support for online scaling of the reserved capacity pool size. Leveraging the Polaris session architecture, Azure Synapse is unique in supporting both serverless and reserved pools with online scaling; the pool form factor represents the next generation of the current Azure SQL DW service, which is subsumed as part of Synapse. The same data can simultaneously be operated on from both serverless SQL and SQL pools.
- *Distributed cost-based query optimization over the data lake.* Related systems such as Snowflake [11], Presto [17, 18] and LLAP [14] do query optimization, but they have not gone through the years of fine-tuning of SQL Server, whose cost-based selection of distributed execution plans goes back to the Chrysalis project [19]. A novel aspect of Polaris is how it carefully re-factors the optimizer framework in SQL Server and enhances it to be cell-aware, in order to fully leverage the Query Optimizer (QO), which implements a rich set of execution strategies and sophisticated estimation techniques. We discuss Polaris query optimization in Section 5; this is key to the performance reported in Section 10.
- *Massive scale-out of state-of-the-art scale-up query processor.* Polaris has the benefit of building on one of the most sophisticated scale-up implementations in SQL Server, and the scale-out framework is designed expressly to achieve this—tasks at each node are delegated to SQL Server instances—by carefully re-factoring SQL Server code.
- *Global resource-aware scheduling.* The fine-grained representation of tasks across all queries in the Polaris workflow-graph is inspired by big data task graphs [3, 4, 5, 6], and enables much better resource utilization and concurrency than traditional data warehouses. Polaris advances existing big data systems in the flexibility of its task orchestration framework, and in maintaining a global view of multiple queries to do resource-aware cross-query scheduling. This improves both resource utilization and concurrency. In future, we plan to build on this global view with autonomous workload management features. See Section 6.
- *Multi-layered data caching model.* Hive LLAP [14] showed the value of caching and pre-fetching of column store data for big data workloads. Caching is especially important in cloud-native architectures that separate state from compute (Section 2), and Polaris similarly leverages SQL Server buffer pools and SSD caching. Local nodes cache columnar data in buffer pools, complemented by caching of distributed data in SSD caches.

2. SEPARATING COMPUTE AND STATE

Figure 1 shows the evolution of data warehouse architectures over the years, illustrating how state has been coupled with compute.

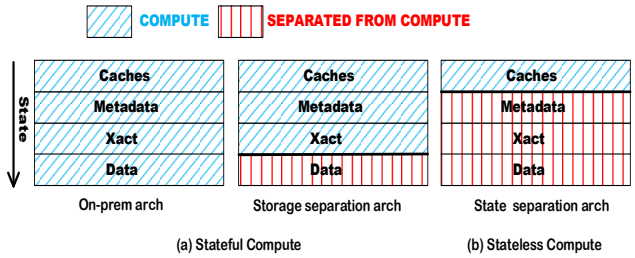


Figure 1. Decoupling state from compute.

To drive the end-to-end life cycle of a SQL statement with transactional guarantees and top tier performance, engines maintain *state*, comprised of *cache*, *metadata*, *transaction logs*, and *data*. On the left side of Figure 1, we see the typical shared-nothing on-premises architecture where all state is in the compute layer. This approach relies on small, highly stable and homogenous clusters with dedicated hardware for Tier-1 performance, and is expensive, hard to maintain, and cluster capacity is bounded by machine sizes because of the fixed topology; hence, it has scalability limits.

The shift to the cloud moves the dial towards the right side of Figure 1 and brings key architectural changes. The first step is the *decoupling of compute and storage*, providing more flexible resource scaling. Compute and storage layers can scale up and down independently adapting to user needs; storage is abundant and cheaper than compute, and not all data needs to be accessed at all times. The user does not need compute to hold all data, and only pays for the compute needed to query a working subset of it.

Decoupling of compute and storage is not, however, the same as *decoupling compute and state*. If any of the remaining state held in compute cannot be reconstructed from external services, then compute remains stateful. In *stateful* architectures, state for in-flight transactions is stored in the compute node and is not hardened into persistent storage until the transaction commits. As such, when a compute node fails, the state of non-committed transactions is lost, and there is no alternative but to fail in-flight transactions. Stateful architectures often also couple metadata describing data distributions and mappings to compute nodes, and thus, a compute node effectively owns responsibility for processing a subset of the data and its ownership cannot be transferred without a cluster re-start. In summary, resilience to compute node failure and elastic assignment of data to compute are not possible in stateful architectures. Several cloud services and on-prem data warehouse architectures fall into this category, including Red Shift, SQL DW, Teradata, Oracle, etc.

Stateless compute architectures require that compute nodes hold no state information, i.e., all data, transactional logs and metadata need to be externalized. This allows the application to partially restart the execution of queries in the event of compute node failures, and to adapt to online changes of the cluster topology without failing in-flight transactions. Caches need to be as close to the compute as possible, and since they can be lazily reconstructed from persisted data they don't necessarily need to be decoupled from compute. Therefore, the coupling of caches and compute does not make the architecture stateful.

Polaris is a cloud-native distributed analytics system that follows a stateless architecture. In the remainder of the paper we go through the technical highlights of the architecture, and finally, we present results of running all 22 TPC-H queries at 1PB scale on Azure.

3. THE POLARIS DATA ABSTRACTION

A key objective for Polaris is to be a scale-out query engine for relational data as well as heterogeneous datasets stored in distributed file systems such as HDFS. The Polaris data model is therefore designed with the following considerations in mind:

- **Abstraction from the data format**. Polaris, as an analytical query engine over the data lake, must be able to query any data, relational or unstructured, whether in a transactionally updatable managed store or an unmanaged file system. Hence, we need a clean abstraction over the underlying data type and format, capturing just what's needed for efficiently parallelizing data processing. A dataset in Polaris is logically abstracted as a collection of *cells* that can be arbitrarily assigned to compute nodes to achieve parallelism. The Polaris distributed query processing framework (DQP), operates at the cell level and is agnostic to the details of the data within a cell. Data extraction from a cell is the responsibility of the (single node) query execution engine, which is primarily SQL Server, and is extensible for new data types.

- **Wide distribution**. For scale-out processing, each dataset must be distributed across thousands of buckets, or subsets of data objects, such that they can be processed in parallel across nodes. In Polaris, this can be expressed as the requirement that a dataset must be uniformly distributed across a large number of cells.

3.1 Data Cells

As shown in Figure 2, a collection (e.g., table) of data objects (e.g., rows) in Polaris can be logically abstracted as a collection of *cells Cij* containing all objects *r* such that $p(r) = i$ and $h(r) = j$.

The hash-distribution h(r) is a system-defined function applied to (a user-defined composite key *c* of) *r* that returns the hash bucket number, or *distribution*, that *r* belongs to. The hash-distribution *h* is used to map cells to compute nodes, and the system chooses *h* to hash datasets across a large number of buckets so that cells (and thus, computation) can be distributed across as many compute nodes as needed. Further, computationally expensive operations such as joins and vector aggregation can be performed at the cell level without incurring data movement if either the join keys or grouping keys are aligned on the hash-distribution key.

The partitioning function p(r) is a user-defined function that takes as input an object *r* and returns the *partition i* in which *r* is positioned. This is useful for aggressive partition pruning when range or equality predicates are defined over the partitioning key. (If the user does not specify p for a dataset, the partition pruning optimization is not applicable.)

Cells can be grouped physically in storage however we choose (examples of groupings are shown as dotted rectangles in Figure 2), so long as we can efficiently access *Cij*. Queries can selectively reference either cell dimension or even individual cells depending on predicates and type of operations present in the query.

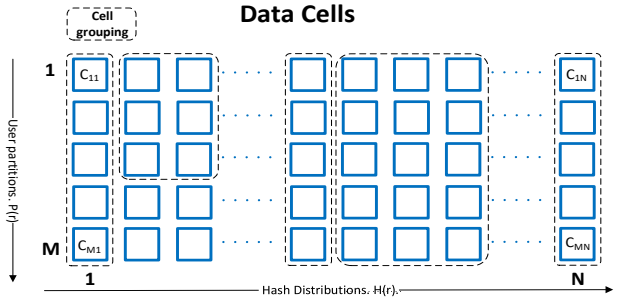


Figure 2. Polaris Data Model

Flexible Assignment of Cells to Compute

Query processing across thousands of machines requires query resilience to node failures. For this, the data model needs to support a *flexible allocation* of cells to compute, such that upon node failure or topology change, we can re-assign cells of the lost node to the remainder of the topology. This flexible assignment of cells to compute is ensured by maintaining metadata state (specifically, the assignment of cells to compute nodes at any given time) in a durable manner outside the compute nodes.

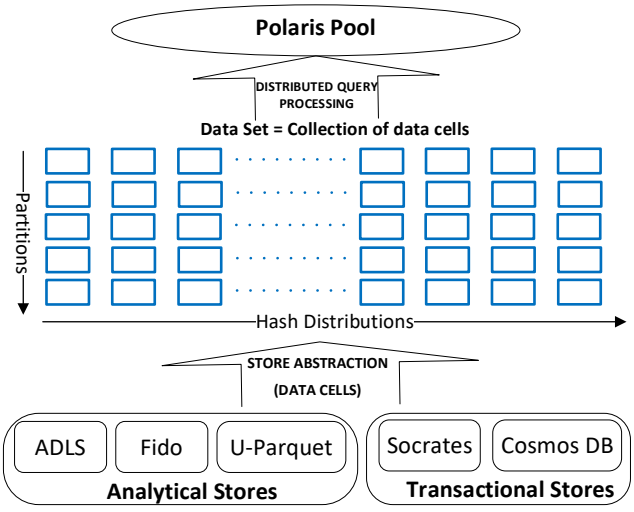


Figure 3. Store Abstraction via Data Cells

Storage Abstraction

Polaris abstracts distributed query processing from the underlying store via data cells. As shown in Figure 3, any dataset can be mapped to a collection of cells, which allows Polaris to do distributed query processing over data in diverse formats, and in any underlying store, as long as efficient access to individual cells is provided by the storage server. As such, Polaris can perform

highly scalable distributed query processing over analytical stores such as ADLS [20], Fido [2], and Delta [21], as well as transactional stores such as Socrates [22] and Cosmos DB [23]. Of course, when data is stored in columnar formats tailored for vectorized processing, this further improves relational query performance.

A Note on Queries

In this paper, we mostly focus on relational queries (with the exception of Section 10.4). Data objects are assumed to have attributes required by relational operators to which they are input. That said, the generality of the data abstraction underlying Polaris's query processing means that we can handle datasets represented in diverse formats and stored in different repositories. For example, Polaris can run directly over data in HDFS and in managed transactional stores. Further, different objects in a dataset could differ in the attributes attached to them, and objects could have additional uninterpreted attributes.

4. MAPPING CELLS TO COMPUTE

A fundamental aspect in distributed execution is how we map cells (of source datasets as well as intermediate results) to compute nodes for various operations involved in the execution of a query. As noted above, we map cells to nodes using the hash-distribution *h*. We now discuss this in more detail.

4.1 Distribution Properties

As discussed above, data objects (e.g., tuples or rows) in a cell are hash aligned, i.e., if *c* is the composite key, all objects that hash to the same cell have the same hash value or *distribution h(c)*. Further, if two objects hash to different distribution values, they must differ on the composite key *c*. As degenerate cases, objects may be distributed round-robin or mapped to a single cell. We introduce the following notation for how objects in a dataset are hashed (or not) across cells:

1. $h[c]$: Objects in a dataset P are mapped to cells using a hash-distribution on column c. Also denoted as: $P^{[c]}$.
2. All objects in the dataset are hashed to the same value, i.e., there is a single hash-bucket: P^{1}
3. Objects in dataset P are not hash-distributed across cells; this situation arises sometimes for intermediate results. Also denoted as: $P^{\emptyset}$

The above *distribution properties* are used by the Polaris Distributed Query Optimizer (DQO) for two fundamental purposes: (1) to guarantee functional correctness of parallel execution of operations such as joins and vector aggregations, and (2) they are used as *interesting properties* by the DQO while enumerating physical distributed alternatives in the search space.

Distribution Properties as Correctness Filters

The input distribution properties of a relational operator are used to guarantee functional correctness when enumerating the physical execution alternatives across multiple compute nodes. For instance, an inner join requires both of its inputs to be hash aligned on the join column, or one input to be mapped to a single hash-bucket, in order to return the correct results while operating only on input cells available locally at each node:

$$P \bowtie^{a=b} Q: \{\{P^{[a]} \wedge Q^{[b]}\} \vee \{P^{1}\} \vee \{Q^{1}\}\}$$

We refer to such correctness criteria on inputs as *required distribution properties*. During the enumeration of the alternative physical distributed plans in the search space, the DQO uses required distribution properties on operators to discard alternatives. The list of required properties for each relational algebra operation is listed in the appendix of this paper.

Distribution Properties as "Interesting Properties"

System R [24] introduced the concept of *interesting properties*, namely physical properties (e.g., sort order) such that the best plan for producing (intermediate) tables with each interesting property is saved during the enumeration of the search space. Thus, the cheapest plan for producing an intermediate table in sorted order by the first column would be saved even if there is a cheaper plan to produce the same table unsorted or in a different sort order. Similarly, in the distributed search space, the Polaris DQO uses the required distribution properties of relational algebra operators as interesting properties. When enumerating the physical plan alternatives bottom-up, the best plan for each property and the best plan overall based on cost are kept.

4.2 Data Move Enforcers

Polaris provides physical operators called *data move enforcers* that can read data from a source dataset and produce a target dataset with different distribution properties:

- *Hash operator*, $\boldsymbol{H_d}$. Re-distributes every object (in every cell of the dataset) by hashing on column *d*. The number of cells in the output dataset can differ from the input.

$$H_d(P^{[c]}) = P^{[d]}$$
$$H_d(P^{1}) = P^{[d]}$$
$$H_d(P^{\emptyset}) = P^{[d]}$$

- *Broadcast operator*, $\boldsymbol{B}$. Maps the input dataset to a single cell and replicates it across multiple locations.

$$B(P^{[c]}) = P^{1}$$
$$B(P^{\emptyset}) = P^{1}$$

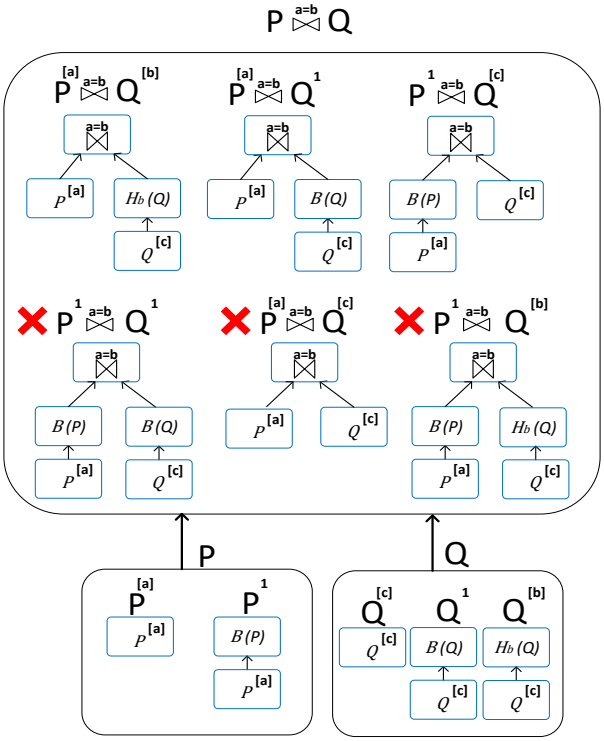


Figure 4. Enumeration of the search space for inner join

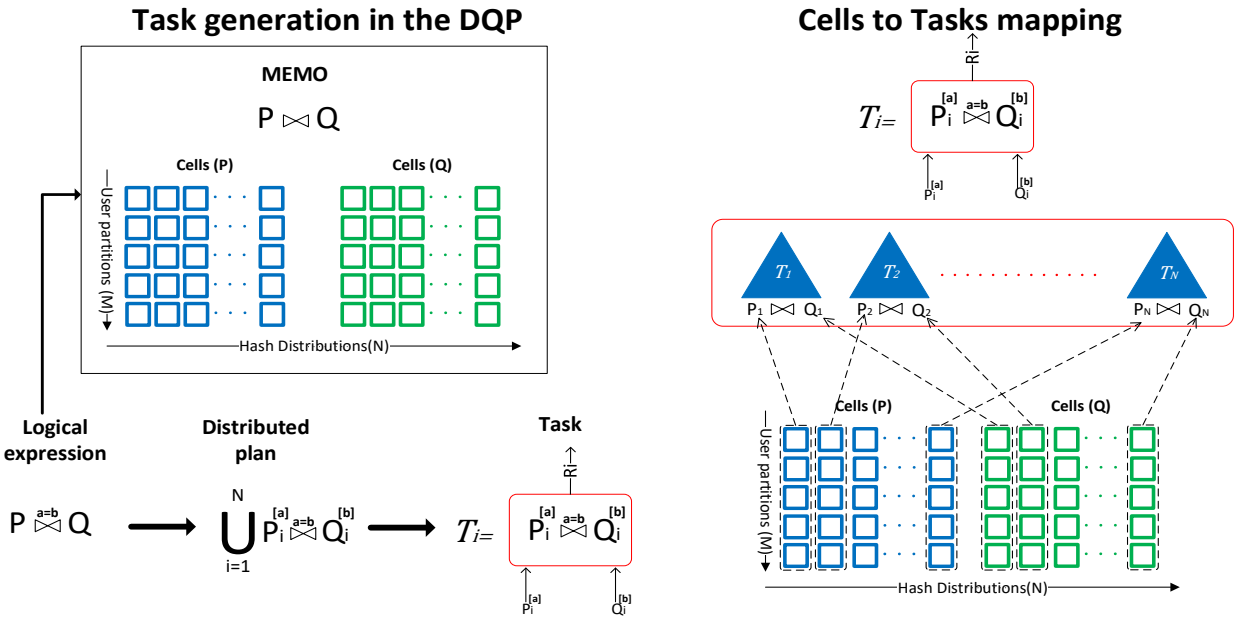


Figure 5. Execution Model

As an example, Figure 4 shows the enumeration of the alternative distributed physical execution plans for an inner join, $P \bowtie^{a=b} Q$ where P and Q are (say, files in a data lake or tables in a managed distributed relational store) hashed on a and c respectively ($P^{[a]}$ and $Q^{[c]}$). The enumeration of physical alternatives starts with the scans of P and Q, shown in the bottom-most part of the figure. Q is hash distributed on column *c*, hence, $Q^{[c]}$ is the first alternative generated. Replication and hash distribution on *b* are interesting properties pushed top-down, leading to the enumeration of sub-plans Q^1 and $Q^{[c]}$ respectively. P is hash distributed on column *a*, generating $P^{[a]}$ as the first alternative. Replication and hash distribution on *a* are also interesting properties pushed top-down. Since we already satisfy hash distribution on *a* via $P^{[a]}$, we only need to produce P^1. The plan node in the top half of Figure 4 shows the enumeration of plans for the join operation; this is a permutation of the alternatives produced by its children at the bottom of the figure. During the enumeration, correctness filters are applied, thereby eliminating $P^{[a]} \bowtie^{a=b} Q^{[c]}$ from the search space, since it does not satisfy any of the distribution properties required by an inner join. For the remaining alternatives, only the best plan for each interesting property is kept:

$$P^{[a]} \wedge Q^{[b]} : P^{[a]} \bowtie^{a=b} Q^{[b]}$$

$$P^1 : P^1 \bowtie^{a=b} Q^{[c]}$$

$$Q^1 : P^{[a]} \bowtie^{a=b} Q^1$$

Finally, the best distributed query plan will be chosen based on the cheapest of the three options. Data move enforcers are expensive operators due to the cost of data re-distribution; hence, the cheapest plan is the one that minimizes data movement, as explained in [19].

5. FROM QUERIES TO TASK DAGS

A fundamentally new aspect of Polaris is its fine-grained representation and tracking of query execution. In this section, we describe how a query is compiled and optimized into an executable DAG of tasks that correspond to units of distributed execution.

5.1 Polaris Tasks

A key challenge in Polaris was how to essentially re-architect distributed query processing while leveraging as much of existing SQL Server capabilities as possible and ensuring that the resulting system was a faithful implementation of all user-visible semantics.

To this end, all incoming queries in Polaris are compiled in two phases. The first phase of the compilation stage leverages SQL Server Cascades QO to generate the logical search space, or MEMO [25, 26]. The MEMO contains all logical equivalent alternative plans to execute the query. A second phase performs distributed cost-based QO to enumerate all physical distributed implementations of these logical plans and picks one with the least estimated cost. The outcome is a good distributed query plan that takes data movement cost into account, as explained in [19].

When enumerating the physical space during the second phase of the QO process, a query plan in the MEMO is seen as a directed acyclic graph (DAG) of physical operators, each corresponding to an algebraic sub-expression *E* in the query. For simplicity, we use *E* to denote both the expression and its instantiation as an operator in the MEMO. Operator *E* has a degree of partitioned parallelism *N* that defines the number of instances of *E* that run in parallel, each on a partition of the input. We denote the distributed execution of *E* as $\bigcup_{i=1}^{N} E_i$, where E_i represents the execution of *E* over the i^{th} hash-distribution of its inputs, and *N* is the degree of parallelism.

We illustrate the notation by means of an example. Figure 5 depicts an expression that consists of a hash aligned join between two input relations, P and Q. As shown on the left, the cell representation of user files over the lake is captured during MEMO generation by SQL Server—the first stage of QO pulls metadata from external services such as remote meta-stores that contain information on the collection of files/tables, partitions and distributions.

For this example, the input data cells are *N*-way hash-distributed such that the parallel distributed query plan is represented through the union of the join operation on each hash-distribution pair; (in contrast to the example of the previous section) P and Q are already hash-aligned on the join column, satisfying the required

distribution properties of the join operator. The same notation can be extended to represent more complex relational expressions and distribution variations, but we omit the details.

Next, we introduce the notion of a ***task*** T_i as the physical execution of an operator *E* on the i[th] hash-distribution of its inputs. Tasks are instantiated templates of (the code executing) expression *E* that run in parallel across *N* hash-distributions of the inputs, as illustrated in Figure 5 with blue triangles. A task has three components:

- *Inputs.* Collections of cells for each input's data partition. These cells can be stored either in highly available remote storage, or in temporary local disks.
- *Task template*. Code to execute on the compute nodes, representing the operator expression *E*.
- *Output.* Output dataset represented as a collection of cells produced by the task. The output of a task is either an intermediate result for another task to consume or the final results to return to the user, and is distributed across several nodes corresponding to the *consuming* task's degree of parallelism.

5.2 The Query Task DAG

In general, the distributed query plan is represented as a directed acyclic graph (DAG) (of operators or tasks) rather than a single node to capture the structure of sub-expressions in the query, including data-flow dependencies and required distribution properties of corresponding operators.

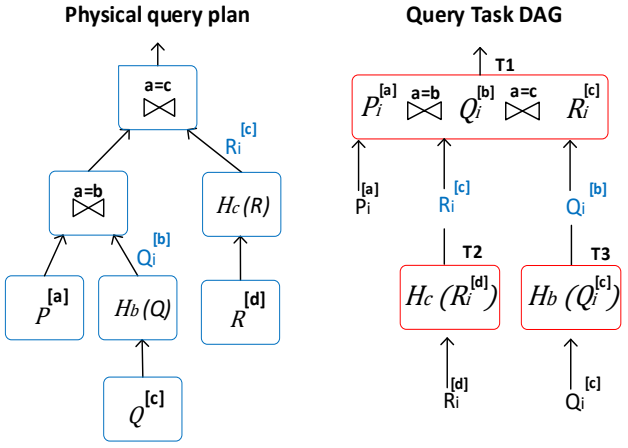


Figure 6. The Query Task DAG

Each vertex contains an operator corresponding to an expression *E* in the query and has a corresponding task template, instantiated across multiple nodes over hash-distributions of the inputs for the vertex. Edges represent dataflow dependencies, and if the consuming vertex *E* does not support pipelining, induce precedence constraints over "consumer tasks" created by instantiating *E* across compute nodes over hash-distributed inputs of *E*. That is, "consumer" tasks cannot start until the corresponding tasks of the producer vertexes of the edge have completed.

Precedence constraints are inherently blocking and define changes of the distribution properties of the data cells consumed by parent tasks. As explained earlier, the DQO injects changes of distribution properties via data move enforcers to achieve correctness, or a better distributed alternative plan to speed up query execution. Therefore, the subtree of physical operators rooted on a move enforcer defines the input and output boundaries of a task. Data move enforcers are blocking operators such that all their output data cells are persisted in local storage before they can be processed by the consumer task.

Tasks in the DAG without precedence constraints can execute in parallel, thereby achieving independent parallelism between different tasks of a query. Figure 6 expands on the example in Figure 4 with an additional join. The left hand side of the figure illustrates the physical distributed query plan that has two move enforcers such that the join between the three relations are hash aligned into a final task, resulting into a query DAG with a total of three tasks.

5.3 SQL Server Scale-up for Task Execution

The example in Figure 5 also illustrates an additional optimization carried out in the second phase of cost-based distributed query optimization. Observe how vertexes in the MEMO corresponding to two join operators have been combined into a single vertex that carries out both joins—this is because all three input datasets (*P*, *Q*, and *R*) are hash aligned on the same column by the preceding move enforcer operations. Thus, in general, the template for a task can include code for an algebra expression involving multiple operators.

While we could perform the three-way join in this example in two sequential tasks, we intentionally seek to make tasks be maximal units of work. This allows us to more effectively leverage the sophisticated scale-up columnar query processor in SQL Server. At each compute node, the task template of the algebraic expression *E* corresponding to the task is encoded back into T-SQL and executed natively in SQL Server. In this approach, the blocking nature of the boundaries of a task actually help SQL Server to optimize the template code of a task with fresh stats from intermediate inputs.

6. TASK ORCHESTRATION

Arguably the biggest engineering challenge in Polaris is orchestration of tasks.

- The scale is daunting—the amount of data could be petabytes, leading to millions of cells; the number of compute nodes used in a single query could be in the thousands; and the number of tasks could be in the millions.
- Execution must be robust to transient failures of nodes, network, storage, and other components (e.g., metadata micro-services), and must guarantee that all precedence constraints are satisfied, and all distributed decisions have quorum.
- Tasks must be automatically re-startable on any node, for auto-scaling and fault-tolerance.

In Polaris, *we introduce a model of the execution of a query as a novel hierarchical composition of finite state machines*. As explained in previous sections, at run time, a query is transformed into a query task DAG, which consists of a set of tasks with precedence constraints.

We refer to each of the following aspects of a query as an *entity*: the query DAG, the task templates and tasks. A leaf-level task template can be instantiated into tasks on its hash-distributed inputs; in this case, we say that the task template entity is *composed* of the instantiated task entities. A non-leaf task template has precedence constraints on other task templates; in this case, the non-leaf task template entity is *composed* of the entities for the task templates on which it depends. For each entity, we refer to the entities of which it is composed as its *dependencies*.

The execution state of each entity is tracked using an associated *state machine* with a finite set of states and state transitions. The state of an entity is a composition of the state of the entities of which

it is composed. States can be either composite or simple. *Simple states* are used to denote success, failure, or readiness of a task template. *Composite states* denote (1) an instantiated task template, or (2) a blocked task template. (Note that an instantiated task will succeed or fail but cannot be blocked; tasks are only instantiated when their inputs are ready.)

A *composite state* differs from a *simple state* in that its transition to another state is defined by the result of the execution of its dependencies. It has a collection of *peer states*, one for each dependency, and a *termination policy intent* aggregates meta-data on execution of dependencies and captures how to interpret the outcome of dependencies, and how to act on other peer states.

The Polaris state machine through its hierarchical state machine composition captures the *execution intent* and it is in this aspect that it differs from other distributed query engines. In other DAG execution frameworks [5, 6, 14], composition is inherent in the execution. In Polaris, the state machine provides a template that is used to orchestrate the execution. The advantage it offers is the ability to formalize how we recover from failures and use the state machine recorder (a log) to observe and reply execution history. Further, for a given a set of workloads in the system, the execution history combined with the rules governing legal transitions can be used to reorder workload executions and explore different execution sequences by forking and resuming execution from selected points in the recorded history; this is future work.

Figure 7 illustrates the entities and state machines for the example in Figure 6. As we can see, the distributed query execution of the query task DAG is modelled as a hierarchical set of state machines. The root query DAG entity starts in the *Run* composite state and instantiates the state machine for the entity corresponding to the (task template T1 representing the) join of *P*, *Q* and *R*. This state machine starts in a (composite) *Blocked* state because it has dependencies on the entities corresponding to (task templates T2 and T3 for) the move enforcers on *Q* and *R*; these task templates are now placed in the scheduler queue. Their state is initialized to *Ready* since they have no dependencies, and they are eventually picked to run by the scheduler.

The state machines for task templates T2 and T3 are instantiated and initialized to the *Run* state. This in turn instantiates tasks for the task templates. If any of these tasks fail, their state machine transitions to the *Failed* state, the failure is detected and the failed task is restarted automatically if the reason is a transient failure (as indicated by the task state machine transition in Figure 7); otherwise the parent state machine retries at a coarser granularity. The state for T2 and T3 becomes *Success* when all its task dependencies succeed. When both move enforcer entities succeed, the root entity T1 is unblocked and placed in the scheduler queue. When it is picked to run, i.e., becomes active, and it is instantiated as join tasks on the hash-partitioned inputs.

In more detail, a state machine in *Failure* triggers an analysis of the type of failure for all dependencies that we classify as retriable, e.g., transient failures caused by node failure. If retriable, then it can transition back to *Blocked*, otherwise, the state machine with *Failure* returns control to the state machine for its parent, which will try to re-schedule execution using additional resources or in turn propagate the failure up the control chain. This is an example of how, in contrast to other systems such as [10, 15, 18], Polaris orchestration gives us flexibility in handling different types of failures by allowing us to specify behavior on termination of a composite state.

To summarize, when in *Ready* state, a task template waits in the queue for the scheduler to pick its turn to execute, then it transitions to *Run*. This is when task entities are instantiated, and the task's state machines are executed. The task template transition from *Run* to a terminating state (*Failed* or *Success*), depends on the resulting execution of the instantiated tasks. Note that any entity can transition from *Failed* to *Run* if the failure is transient. The failure is propagated to higher entities only if it is deemed not retriable within the entity's state machine.

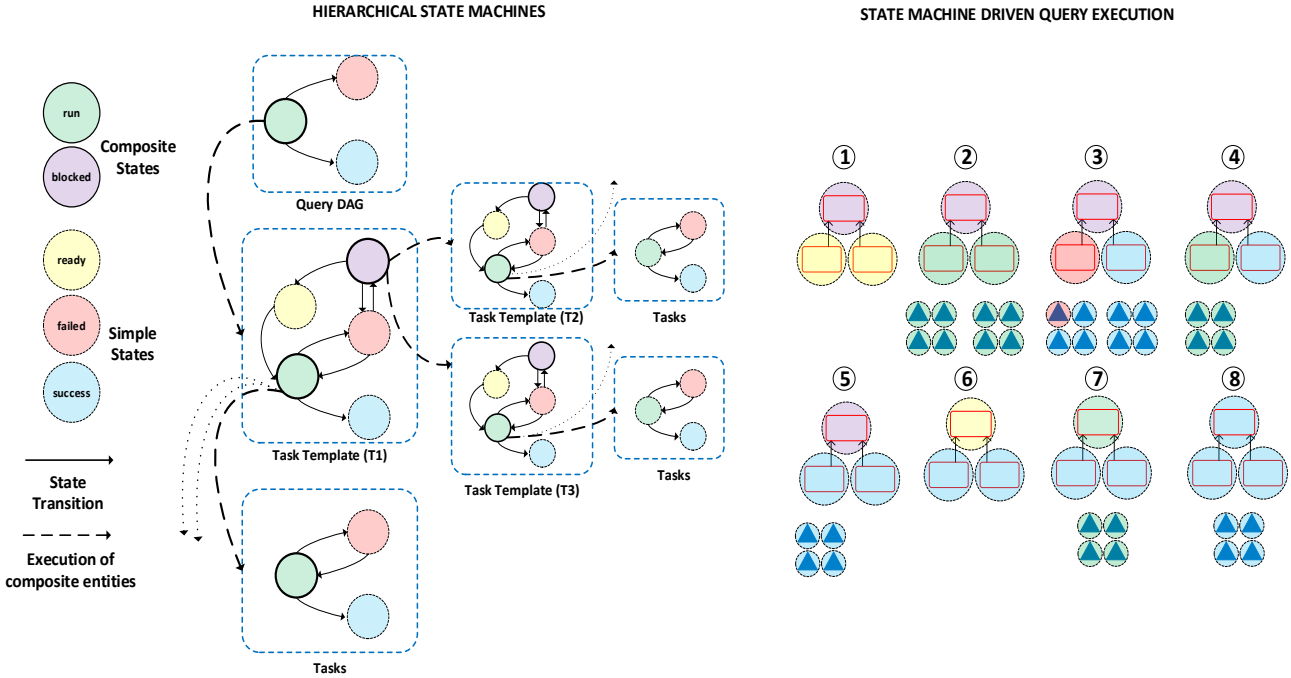


Figure 7. Hierarchical composition of state machines for distributed query execution

Modelling the distributed query execution of queries via hierarchical state machines has the following goals:

- **Satisfy precedence constraints**. The execution of the query task DAG is carried out top-down in a topological sort order such that every task with precedence constrains is blocked on completion of its input tasks. For example, as shown in the right-hand side of Figure 7, the root task is blocked (Step 1) until its two dependencies are completed (Step 6).
- **Reliable execution**. We use the state machines to have fine grain control at task level and define a predictable model for recovering from failures. Completion and failure propagation are done bottom-up using the compositional nature of states. In Step 3 illustrates a case where on container failure during the execution of a task, the error propagates to the parent task template, which retries its execution.
- **Reproducibility at scale**. States and transitions are logged by all entities. This allows for predictability and reproducibility regardless of the complexity of the workload and the scale. This is also a fundamental building block for debugging and resumable execution upon failover.
- **Concurrency**. Fine grain control at large scale often comes with large memory requirements and thread contention due to many subroutines running concurrently. Hierarchical state machines allow us to track the state of all entities in the workload with a low memory overhead: there is only one state machine for a task entity and all instantiations run through its states and transitions. Also, the Polaris query processor has been built from scratch using .NET's task asynchronous programming model to eliminate the need for blocking synchronization primitives across subroutines, thus minimizing thread contention and maximizing OS thread utilization. The gains are seen in Section 10.2.

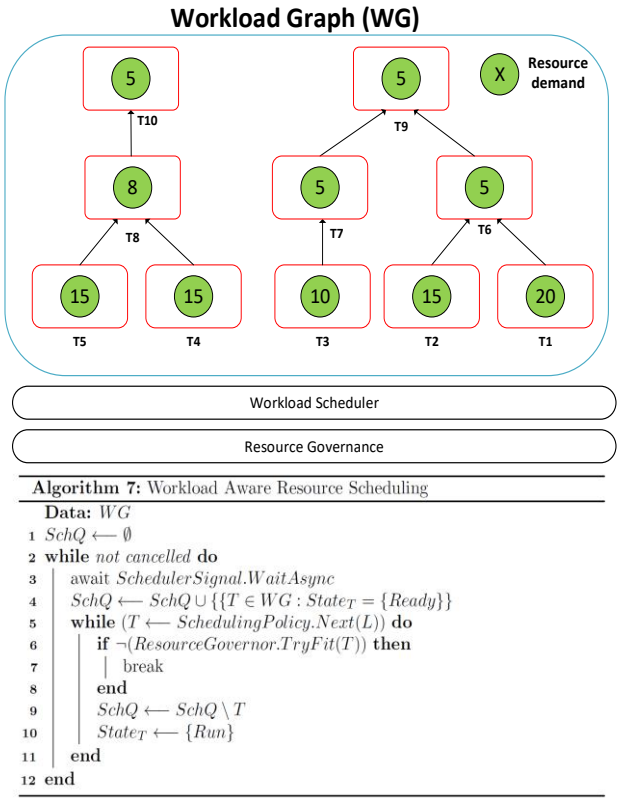
7. WORKLOAD AWARE SCHEDULING

Polaris must handle highly concurrent workloads ranging from dashboarding scenarios running thousands of light weight queries, to reporting scenarios executing a set of highly complex analytical queries. There are potentially millions of tasks to be orchestrated for execution by the Polaris DQP. In the previous section we described how hierarchical state machines enable us to efficiently handle distributed task orchestration at very large scale. In this section we cover how Polaris schedules tasks for high concurrency.

Task scheduling in Polaris is based on a global view of all active queries called the *workload graph*, generalizing the representation of a single query as a DAG of tasks to represent the entire workload by combining task DAGs of all active queries.

Each task in the workload graph has an associated *resource demand* that is an extension of the model in Ganguly [27] to d-dimensional preemptable resources proposed in [28, 29]. We define a d-dimensional resource vector that has time and space shared constraints where each dimension specifies an aspect of resource consumption. Fungible resources such as memory and CPU can be sliced across tasks at a low cost, and each task's requirement for a given resource can be stretched at execution time. On the other hand, more rigid resources such as temp space on local disks must also be satisfied. Stretching temp space across independent tasks is prohibitively expensive since it would require swapping pages in and out from/to remote storage.

The resource demand for each task is computed as a function of inputs and outputs of each physical operator in the template code for the task. Analogously, Polaris also models each compute node as a d-dimensional bin of resources such that placement of tasks to containers is based on policies that can be autonomously tuned based on resource consumption profiles across all nodes.



Algorithm 7: Workload Aware Resource Scheduling

```
Data: WG
SchQ ⟵ ∅
while not cancelled do
    await SchedulerSignal.WaitAsync
    SchQ ⟵ SchQ ∪ {{T ∈ WG : State_T = {Ready}}
    while (T ⟵ SchedulingPolicy.Next(L)) do
        if ¬(ResourceGovernor.TryFit(T)) then
            break
        end
        SchQ ⟵ SchQ \ T
        State_T ⟵ {Run}
    end
end
```

Figure 8. Workload aware resource scheduling algorithm

The representation of the workload as a global graph of tasks with resource demands allows us to redefine the multi-query scheduling problem as a task scheduling problem with precedence constraints: the goal is scheduling d-dimensional tasks on d-dimensional containers to complete in the minimum amount of time possible while ensuring that at all times, we are within all d dimensions of resources available to us. Figure 8 shows the representation of the *workload graph* for two query DAGs. In green circles we represent the resource demand for each task template. For simplicity, in this example we normalize to just one number, and not the multi-dimensional resource vector used in Polaris. The workload scheduler and the resource governor operate on the workload graph.

The pseudocode of the scheduler is shown on the bottom of the figure. The scheduler is asynchronously waiting for work, and when awoken it adds all task templates in the workload graph that are in *Ready* state to the scheduler queue. Task templates are then dequeued in order specified by the scheduling policy. Currently supported policies include (combinations of): *FIFO*, *sorted by resource demand* (min to max or max to min), and *sorted by proximity to the root*. Intuitively, sorting by proximity to the root biases towards tasks from jobs that are closer to completion (so that their shared resources can be released sooner).

For the next task template in order, the resource governor examines each task to be instantiated. If all these tasks fit in their target location (i.e., each task's resource demand can be accommodated given current local capacity), then the task template is removed from the scheduler queue and transitioned to the *Run* state. Otherwise, we break out of the loop and wait for other tasks to complete so the task template can fit. Note that the target location of a task is fixed by data affinity to exploit cache locality. This novel approach to multi-query workload—generalizing task scheduling from big data systems to consider tasks across all active queries—can improve concurrency for the following reasons:

- **A task template is the unit of scheduling**. The scheduling order applies to the task template entity and not a query. A finer grain unit of scheduling allows for better packing strategies, helping maximizing resource utilization.
- **Weighted policies for resource governance**. The placement of the task in the target compute server is based on resource fit to maximize load while avoiding over-provisioning. For this we use a weighted policy to pack tasks into the compute capacity available at a node. The policy has two variations, one that caps the amount of resources that can be granted to a task, and another one that does not. If the task does not fit in the available compute, it is put back into the queue till tasks complete and capacity is freed.
- **Increased flexibility in task ordering**. Scheduling policies define the order in which tasks are executed as they become ready for execution. By looking at ready tasks across all queries, taking into account resource pressure in the system, we are able to pick orderings that would not be permissible otherwise. For instance, consider the example in Figure 8, applying a max to min scheduling policy. The scheduler queue SchQ starts with {T1, T2, T3, T4, T5} with scheduling order {T1, T2, T4, T5, T3}. As we go through the loop, T3 does not fit, so only four out of the five task templates transition to *Run* state. Next all T4 and T5 complete and the scheduler is awoken. SchQ now contains {T8, T3} with scheduling order {T3, T8}. Now, if the workload manager detects pressure in the system because of disk resources held by previously completed task templates, it can choose to swap the scheduling policy to sort by proximity to the root to release pressure in the system. In this example, the scheduling order would change to be {T8, T3}. The study of scheduling and resource management policies to consider SLAs and avoid starvation is out of scope for this paper and will be addressed in future work.
- **Resource driven query admission control**. Back pressure can be driven by a ratio of capacity (demand vs. available). Concurrency is only limited by available capacity, and the admission of a query is only denied when we cannot guarantee SLAs due to a capacity crunch.

8. SERVICE ARCHITECTURE

Figure 9 illustrates the architecture of the query service for two Polaris pools sharing the same centralized (metadata and transaction) services. There are two important aspects to note:

- **Stateless architecture within a pool**. The Polaris architecture falls into the stateless service architecture from Figure 1.b, as discussed in Section 2. All services within a pool are stateless: (i) data is stored durably in remote storage, and is abstracted via data cells, and (ii) the metadata and transactional log state is off-loaded to centralized services. (We do not go into the architecture of the centralized services in detail; briefly, they are built for HA and performance using Azure SQL DB.)
- **Multiple-pools**. Placing the state in centralized services coupled with a stateless micro-service architecture within a pool means multiple compute pools can transactionally access the same logical database.

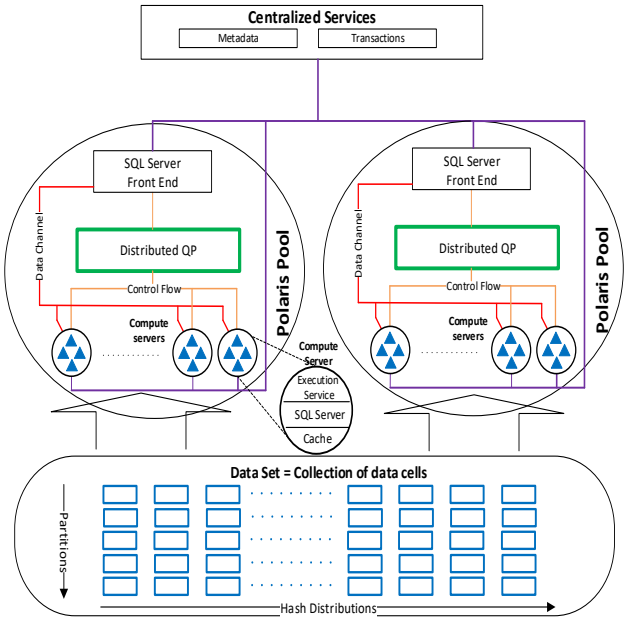


Figure 9. Polaris service architecture

8.1 Stateless micro-service architecture

A Polaris pool consists of a set of micro-services each with well-defined responsibilities. The SQL Server Front End (SQL-FE) is the service responsible for compilation, authorization, authentication, and metadata. Metadata is used by the compiler to generate the search space (the MEMO) for incoming queries and bind metadata to data cells. The Distributed Query Processor (DQP) is responsible for distributed query optimization, distributed query execution, query execution topology management and workload management (WLM). Finally, a Polaris pool consists of a set of compute servers that are, simply, an abstraction of a host provided by the compute fabric, each with a dedicated set of resources (disk, CPU and memory). Each compute server runs two micro-services: (a) an Execution Service (ES) that is responsible for tracking the life span of tasks assigned to a compute container by the DQP, and (b) a SQL Server instance that is used as the back-bone for execution of the template query for a given task and holding a cache on top of local SSDs (in addition to in-memory caching of hot data). Data can be transferred from one compute server to another via dedicated data channels. The data channel is also used by the compute servers to send results to the SQL FE that returns the results to the user. The life cycle of a query is tracked via control flow channels from the SQL FE to the DQP, and the DQP to the ES.

As explained in Section 2, no essential state is held by any micro-service in Polaris. While caches are stored by the compute servers, upon fail-over, they can be easily re-constructed.

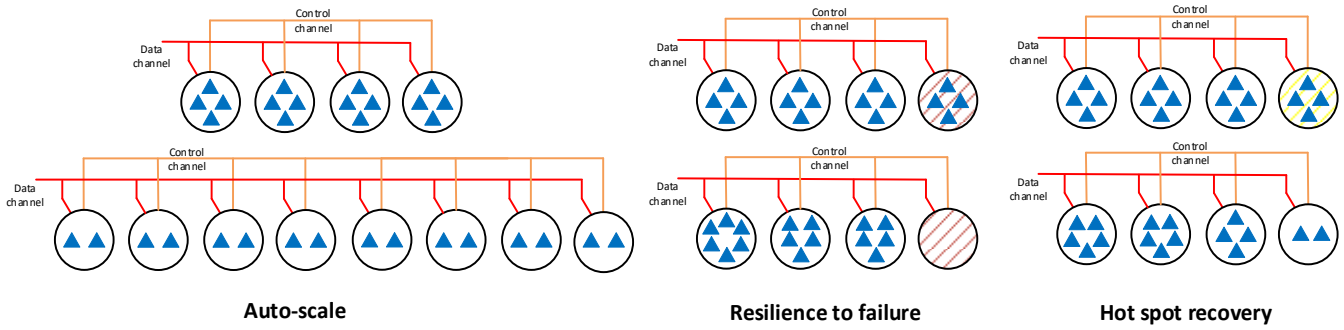


Figure 10. Elastic Compute Scenarios

8.2 Service form factors

The separation of state and compute coupled with the auto-scaling capabilities of a pool (explained in the next section) allow us to support very high concurrency levels within each pool, as well as enabling all of the following user-facing service form-factors:

- **Serverless**. One system managed Polaris pool with auto-scale ranging from 0 to N compute nodes where N is constrained only by capacity within Azure compute.
- **Capacity reservation**. A dedicated Polaris pool with a minimum reservation of capacity and auto-scale capacity up to a maximum user-specified size.
- **Multiple pools**. Multiple Polaris pools with capacity reservation. Pool sizes can either be defined by the user or they can grow and shrink dynamically.

9. ELASTIC QUERY PROCESSING

The infrastructure of a cloud is inherently elastic in that compute containers (e.g., VMs, k8 containers) can be obtained or released nearly instantly. This means nodes can be added and removed from a query processing compute topology in a matter of seconds. With appropriate telemetry, the system can auto-scale up or down proactively based on workload needs or react to unexpected events such as faulty nodes and infrastructure upgrades.

In any of these scenarios, the Polaris query processor must ensure that tasks can be flexibly assigned to compute nodes in dynamically changing query execution topologies. We achieve this objective by leveraging several aspects of the Polaris framework:

- Separation of state and compute.
- Flexible abstraction of datasets as cells.
- Task inputs defined in terms of cells.
- Fine-grained orchestration of tasks using state machines.

Figure 10 depicts examples of key scenarios we unlock with this architecture. We explain each one in the following Sections.

9.1 Auto-Scale

The Polaris DQP requests the underlying compute fabric for more containers to adjust to peaks in the workload and re-distributes tasks to transparently leverage the new containers. Note that in-flight tasks in the previous topology continue running, while new queries get the new compute power with appropriate load balancing. In Figure 10, we show a doubling of compute capacity; however, we can add capacity in increments of just one node.

The Polaris DQP also can autonomously scale down the compute node topology (in increments of one or more nodes) when utilization drops sufficiently.

Resilience to Node Failures

Figure 10 also illustrates how the Polaris DQP recovers from node failures while tasks are running. If a server fails, the DQP rebalances the tasks in the failed node across the rest of the healthy topology. The fault tolerance model is built into the hierarchical state machine discussed in Section 6. A node failure transitions execution tasks in a container into the *Failed* state. Then the parent task template state machine reacts appropriately—tasks previously assigned to the faulty node are restarted on healthy nodes. This feature is essential for scaling to very large queries, since the probability of node failure increases with the number of nodes involved.

9.2 Skewed Computations

Figure 10 shows how skewed computation or hot spots are handled. The Polaris DQP and the ES in the compute servers implement a feedback loop that tracks the life span of execution tasks on a node. If the DQP detects that a node is overloaded (e.g., the yellow node in the figure), it can decide to re-schedule a subset of the tasks assigned to that compute node amongst other nodes where the load is less. If this does not mitigate the hot spot, we fall back on the auto-scale feature to add more nodes to the topology and rebalance the load appropriately. Skewed computations are handled using runtime feedback loops, and our query optimizer does not currently take data skew into account. How to handle skew during query optimization is future work.

9.3 Affinitizing Tasks to Compute

As explained in Section 8, the SQL Server service in the compute server extends caching of hot data to its local SSDs. Accessing data from remote storage is an expensive operation, and therefore the elastic features of the Polaris DQP try to minimize the impact on the cache by consistent assignment of data cells to tasks, and our scheduler assigning tasks to compute based on data collocation, thus, preserving caches upon topology changes. In particular, (a) on topology shrinkage, only the caches of the nodes that are no longer part of the topology are lost, and (b) on topology growth, all the caches from the existing topology are preserved, and only the caches for the new nodes need to be populated.

10. PERFORMANCE EVALUATION

10.1 Goals

Our goals are to obtain an understanding of the performance and concurrency characteristics of Polaris on a single pool over structured and non-structured data. For this we break down our experiments into three dimensions.

Concurrency. We want to stress the DQP with a concurrent workload. The global graph in such scenario consists of thousands of tasks from a thousand different queries such that we showcase

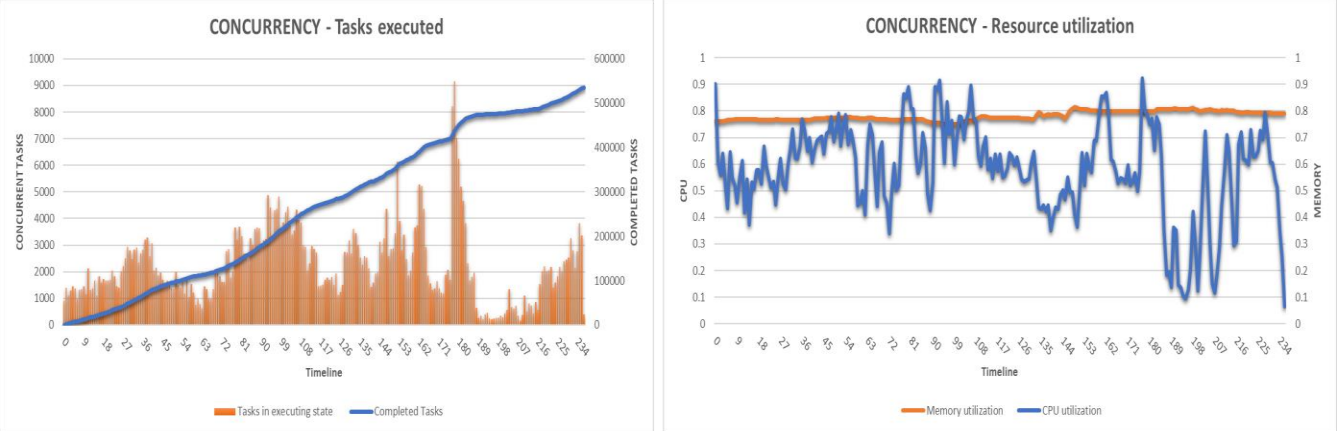


Figure 11. Results for 5k concurrent queries

the resource driven scheduling capabilities of the WLM, and its autonomy around capacity management, access control and resource governance under heavy load. For this we use the TPC-DS [30] workload to run a multi-user environment executing five thousand of queries simultaneously.

Single query performance at PB scale over the data lake. We ran all TPC-H [31] queries at one PB scale across hundreds of machines on Azure public compute. The goal of this experiment is to stress the scalability, elasticity, and fault tolerance capabilities of the service. Note that this is not a validated TPC-H benchmark, the only intent is to demonstrate we can run all queries at a scale that has not been done before.

Querying heterogeneous data. To illustrate that Polaris can run on heterogeneous data, we ran all TPC-H [31] queries at 1 TB scale on a dataset consisting of a variety of data files ranging from raw CSV files to Parquet files with nested attributes. The test was executed using less than 100 cores in Azure public compute. The experiment emphasizes raw file parsing and query optimization capabilities over joins between plain text files and Parquet files with nested attributes.

10.2 Concurrency

The setup

We used the TPC-DS *dbgen* utility to generate a 1TB of raw data and then converted it into parquet files that were stored into Windows Azure Storage Blob (WASB), the Azure Data Lake. Rows do not follow any distribution since we are not focusing on single query performance but stress on concurrency. The application spans five thousand concurrent sessions executing one distinct TPC-DS query each. For this we generated TPC-DS queries 50 times with different predicate ranges and assigned one to each session.

The compute topology

Since the goal of this experiment is to stress the DQP component we choose a rather small compute topology with 10 compute nodes. The hardware configuration of each node consists of 2x20 cores intel processors, 520GB of RAM and 4 SSDs of 1TB each. The network topology is 40Gb throughout; 40Gb NIC, 40Gb TOR, 40Gb CSP.

Results

Figure 11 shows the task execution summary and the resource utilization in the backend nodes. The 5k queries run simultaneously generating a workload graph over 50k task templates that as they are scheduled, they expand to an aggregated total of ~550k instantiated tasks. As task templates are scheduled for execution, tasks are instantiated; the chart on the left of the Figure shows the number of actively executing tasks and aggregated completed tasks at any given point in time for the duration of the test. The chart on the right of the Figure shows the average resource utilization of the compute server for CPU and Memory dimensions. As we can see, we do have a good utilization of the cluster for the duration of the tests. For this experiment we have used *FIFO* scheduling order of task templates, and we think both the resource utilization and time can be improved by using more sophisticated policies; experiments using different scheduling order policies are out of the scope of this paper, and to be carried out in the near future. The main thing to observe is that Polaris is able to handle high concurrency for a complex workload such as TPC-DS packing up to 9k tasks on the 10 compute servers available, and completing approximately 550k tasks.

10.3 Query Performance at Petabyte Scale

The set-up

We used the TPC-H *dbgen* utility to generate a PB of raw data and then converted it into parquet files that were stored in WASB. Parquet files were organized using the data model from Section 3 with both hash partitions and user partitions. The total number of parquet files is ~120k with total compressed size of 360TB.

The compute topology

We deployed a Polaris pool on Azure, consisting of one SQL FE compute instance, one DQP and 420 compute execution services (ES). Each node is a 2x12 cores Intel processor with 192GB of RAM and 4 SSDs of 480GB. The network topology is 40Gb throughout; 40Gb NIC, 40Gb TOR, and 40Gb CSP.

Results

Figure 12 shows execution time for all 22 TPC-H queries at 1PB scale. To the best of our knowledge, this is the first time results have been published at a PB scale. Remarkably, some queries (Q6, Q12, Q15 and Q16) run extremely fast, through partition elimination and distribution alignment of expensive joins, taking advantage of the Polaris data model (Section 3). TPC-H has a few queries that stress the processing limits of any system since they join across all sources with low selectivity and very heavy joins between large dimension and the fact table: Q9 and Q21 are good

examples. Polaris manages to process these queries at PB scale under two hours across 420 machines, demonstrating scalability and resilience.

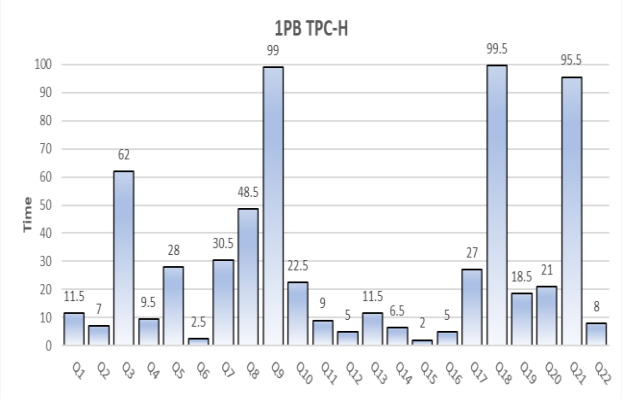


Figure 12. 1PB TPC-H single query performance

10.4 Querying Heterogeneous Data

The set-up

We used the TPC-H *dbgen* utility to generate a TB of raw data in CSV format and then converted the files for the *lineitem*, *customer*, *supplier*, and *nation* tables into Parquet files. Conversion into Parquet for *customer* and *supplier* files was done by organizing contact information (*name*, *address*, *nationkey*, and *phone* columns) as nested types in Parquet. The *lineitem* and *nation* Parquet files were organized with simple types, without nested structure. Files for *orders*, *partsupp*, *part* and *region* were kept in raw CSV format. All files for a single entity were stored in a single folder in WASB.

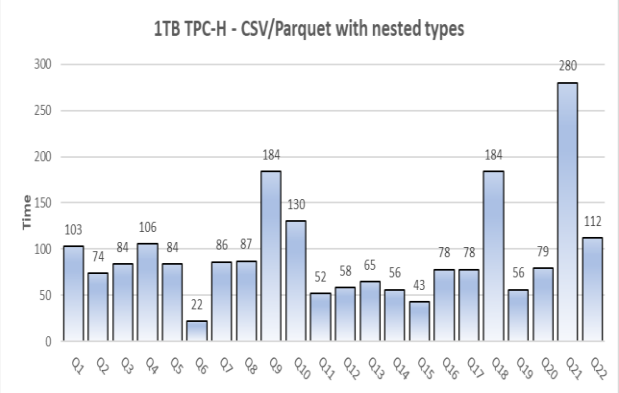


Figure 13. 1TB TPC-H querying heterogeneous data.

The compute topology

We deployed a Polaris pool with one SQL FE compute instance, one DQP and several execution services (ES).

Results

Figure 13 shows the query execution time for all 22 TPC-H queries at 1TB scale that combines querying CSV files for some entities and Parquet files with and without nested types for other entities. Polaris executes all 22 queries and produces good plans even for the most complex queries, which do joins across a variety of files (CSV, Parquet with simple types, and Parquet with nested types). This demonstrates the robustness of the system in handling heterogeneous data sources.

11. Conclusions

In this paper, we presented Polaris, a novel distributed query processing framework in Azure Synapse that seeks to support both big data and relational warehouse workloads, going beyond current systems of either kind in its flexibility and scalability. The architecture is inspired by scale-out techniques from big data systems. It extends these techniques in many ways, notably in the cell abstraction of data, flexible task orchestration framework, and global workload task graph. Polaris also is notable for how it carefully refactors SQL Server's complex codebase in order to leverage its query optimizer and scale-up single-node engine—both of which reflect many years of refinement—while completely rewriting the distributed execution framework.

Polaris is also cloud-native, completely separating compute from both storage and transactional state in order to support agile provisioning and scaling of compute pools. Azure Synapse is unique among cloud services in supporting both serverless and provisioned form factors, with multiple serverless and provisioned SQL sessions able to concurrently operate on the same datasets, across both lake and managed data.

Appendix

Required properties

The following table contains the required properties for the most common algebraic operators. The columns are treated as equivalence classes (transitive closures) when testing the required properties for algebraic correctness. When join predicates have multiple equality conjuncts, correctness holds if hash key of each input is a subset of the columns in the conjuncts from that input. For "Group-By", correctness holds if hash key of input is a subset of the grouping columns. Distributed query processor also supports decomposing aggregations and Top-N into local-global forms, which allows the optimizer to push selective local operators before data movement enforcers. $P^{[a]}$ subsumes $P^{\emptyset}$.

Operator	Required Properties
Inner Join	$P \bowtie^{a=b} Q$: $\{\{P^{[a]} \wedge Q^{[b]}\} \vee \{P^1\} \vee \{Q^1\}\}$
Outer Join	$P \rightarrow^{a=b} Q$: $\{\{P^{[a]} \wedge Q^{[b]}\} \vee \{Q^1\}\}$
Semi-Join	$P \ltimes^{a=b} Q$: $\{\{P^{[a]} \wedge Q^{[b]}\} \vee \{P^1 \wedge Q^{[b]}\} \vee \{Q^1\}\}$
Anti-Join	$P -^{a=b} Q$: $\{\{P^{[a]} \wedge Q^{[b]}\} \vee \{Q^1\}\}$
Group-By	$GB(P, a)$: $\{\{P^{[a]}\} \vee \{P^1\}\}$
Project	$\Pi(P)$: $\{true\}$
Select	$\sigma(P)$: $\{true\}$
Top	$Top(P)$: $\{P^1\}$
Union-All	$P \uplus Q$: $\{\{P^{\emptyset} \wedge Q^{\emptyset}\} \vee \{P^1 \wedge Q^1\}\}$
Union	$P \cup Q$: $\{\{P^{[a]} \wedge Q^{[b]}\} \vee \{P^1 \wedge Q^1\}\}$
Apply	P Apply Q: $\{Q^1\}$

12. REFERENCES

[1]. *Azure Synapse Analytics.* [https://azure.microsoft.com/en-us/services/synapse-analytics/]

[2]. Report, Microsoft. *FIDO: A Cloud-Native Versioned Store With Concurrent Transactional Updates.* 2020.

[3]. Ashish Thusoo et al, *Hive – A Petabyte Scale Data Warehouse Using Hadoop.* Long Beach, California, USA : ICDE Conference, 2010.

[4]. R. Chaiken, et al. *SCOPE: easy and efficient parallel processing of massive data sets.* Auckland, New Zealand : VLDB Conference, 2008.

[5]. Michael et al. *Spark SQL: Relational data processing in Spark.* Armbrust, Melbourne, Victoria, Australia : Proc. ACM SIGMOD, 2015. SIGMOD.

[6]. Michael Isard et al. *Dryad: distributed data-parallel programs from sequential building blocks.* Lisboa, Portugal : Eurosys, 2007.

[7]. Gupta, Anurag et al. *Amazon Redshift and the Case for Simpler Data Warehouses.* Melbourne, Victoria, Australia : SIGMOD Conference, 2015.

[8]. *AWS Athena.* [https://aws.amazon.com/athena/]

[9]. *An Inside Look at Google BigQuery.* [https://cloud.google.com/files/BigQueryTechnicalWP.pdf]

[10]. Melnik, Sergey, et al. *Dremel: Interactive Analysis of Web-Scale Datasets.* s.l. : VLDB Endowment, 2010.

[11]. Dageville, Benoit, et al. *The Snowflake Elastic Data Warehouse.* San Francisco, California, USA : SIGMOD Conference, 2016.

[12]. *Oracle Exadata.* [https://www.oracle.com/technetwork/database/exadata/exadata-technical-whitepaper-134575.pdf]

[13]. *Teradata.* [https://www.teradata.com/]

[14]. *Apache Hive LLAP.* [https://cwiki.apache.org/confluence/display/Hive/LLAP]

[15]. Marcel Kornacker et al. *Impala: A Modern, Open-Source SQL Engine for Hadoop.* Asilomar, California, USA : CIDR, 2015.

[16]. *Presto.* [https://prestodb.io/]

[17]. R. Sethi et al. *Presto: SQL on Everything.* Macao, Macao : ICDE Conference, 2019.

[18]. *Introduction To Presto Cost Based Optimizer.* [https://prestosql.io/blog/2019/07/04/cbo-introduction.html]

[19]. Shankar, Srinath, et al. *Query optimization in microsoft SQL server PDW.* Scottsdale, Arizona, USA : SIGMOD Conference, 2012.

[20]. Ramakrishnan, Raghu et al. *Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics.* Chicago, IL, USA : SIGMOD Conference, 2017.

[21]. *Delta Lake.* [https://delta.io/]

[22]. Antonopoulos, Panagiotis et al. *Socrates: The New SQL Server in the Cloud.* Amsterdam, Netherlands : SIGMOD Conference, 2019.

[23]. Shukla, Dharma et al. *Schema-Agnostic Indexing with Azure DocumentDB.* Kohala Coast, Hawaii : VLDB Conference, 2015.

[24]. Astrahan, Morton M. et al. *System R: relational approach to database management.* s.l. : ACM Transactions on Database Systems, 1976, ACM Transactions on Database Systems, pp. 16-36.

[25]. Graefe, Goetz and McKenna, William J. *The Volcano optimizer generator: extensibility and efficient search.* Vienna, Austria : IEEE International Conference on Data Engineering, 1993.

[26]. Graefe, Goetz. *The Cascades Framework for Query Optimization.* s.l. : Data Engineering Bulletin, 1995, Vol. 18.

[27]. Hochbaum, Dorit S. and Shmoys, David B. *Using dual approximation algorithms for scheduling problems: Theoretical and practical results.* Portland, OR, USA : IEEE, 1985. 26th Annual Symposium on Foundations of Computer Science (scfs 1985).

[28]. Garofalakis, Minos N. and Ioannidis, Yannis E. *Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources.* Athens, Greece : VLDB Conference, 1997. VLDB.

[29]. Garofalakis, Minos N. and Ioannidis, Yannis E. *Multi-dimensional Resource Scheduling for Parallel Queries.* Montreal, Canada : SIGMOD Conference, 1996. SIGMOD.

[30]. The TPC-DS Benchmark. [Online] http://www.tpc.org/tpcds/.

[31]. The TPC-H Benchmark. [Online] http://www.tpc.org/tpch/.