Quill: Efficient, Transferable, and Rich Analytics at Scale

Badrish Chandramouli[†], Raul Castro Fernandez[‡*], Jonathan Goldstein[†],
Ahmed Eldawy[♯*], Abdul Quamar[◦*]

[†]Microsoft Research [‡]MIT [♯]Univ. of Minnesota [◦]Univ. of Maryland

badrishc@microsoft.com, raulcf@csail.mit.edu, jongold@microsoft.com, eldawy@cs.umn.edu, abdul@cs.umd.edu

ABSTRACT

This paper introduces Quill (stands for a *quadrillion tuples per day*), a library and distributed platform for relational and temporal analytics over large datasets in the cloud. Quill exposes a new abstraction for parallel datasets and computation, called *ShardedStreamable*. This abstraction provides the ability to express efficient distributed physical query plans that are transferable, i.e., movable from offline to real-time and vice versa. ShardedStreamable decouples incremental query logic specification, a small but rich set of data movement operations, and keying; this allows Quill to express a broad space of plans with complex querying functionality, while leveraging existing temporal libraries such as Trill. Quill's layered architecture provides a careful separation of responsibilities with independently useful components, while retaining high performance. We built Quill for the cloud, with a master-less design where a language-integrated client library directly communicates and coordinates with cloud workers using off-the-shelf distributed cloud components such as queues. Experiments on up to 400 cloud machines, and on datasets up to 1TB, find Quill to incur low overheads and outperform SparkSQL by up to orders-of-magnitude for temporal and 6× for relational queries, while supporting a rich space of transferable, programmable, and expressive distributed physical query plans.

1. INTRODUCTION

With the growth in data volumes acquired by businesses today, there is a need to deploy rich analytic workflows over the data, that can operate on both historical (bounded) and real-time (unbounded) datasets. Queries in such workflows usually take the form of:

1. *Ad-hoc queries*, one-time queries over the data that often come with the expectation of results at interactive latencies.
2. *Recurring queries*, queries that are carefully authored and deployed to recur periodically, such as daily or hourly reports.
3. *Continuous queries*, queries that execute and incrementally compute results over data as it is received in real-time, and may be back-tested over historical data as well.

Queries are typically issued using a declarative front-end such as SQL (or its temporal dialect), sometimes with integration into the *high-level language* (*HLL*) of the application (e.g., SparkSQL [8]). Queries eventually get deployed as *distributed physical plans* that execute on a multi-node cluster. While ad-hoc queries are well-supported by such a traditional DBMS workflow, our experience with production large-scale uses of Trill [15], a temporal analytics library used across Microsoft, indicates that customers who author large-scale continuous queries (on real-time data and offline data for back-testing) or recurring queries (on offline data) have a unique combination of new requirements. Consider an example scenario:

Example (Ad Platform)**.** *Consider an advertising (ad) platform that tracks user activity such as ads shown and clicks on the ads.*

(1) ***Application A*** *may wish to compute an hourly report of the per-ad count of user activity over a historical dataset, in a cluster of multi-core machines holding fragments of the dataset (one per core), using a combination of per-core and per-machine aggregation and shuffling data by key. Fig. 1 shows some strategies; in Fig. 1(a), we globally shuffle the dataset so that each core in the system has a partition (by AdId) of the dataset, and then perform a per-core aggregation. This can be appropriate if aggregation is very expensive. On the other hand, if we have very few groups (AdId values), Fig. 1(b) first computes an independent partial aggregate per-core, and then shuffles the aggregated results across the cluster (the other strategies, and more complex examples are covered in §3).*

(2) ***Application B*** *may want the same per-ad count query to instead produce an incrementally maintained dashboard of top ads in the last* 5 *minutes, updated every minute. Developers may also need to back-test the query on varying amounts of offline logs at different scales to tune the window size or other (e.g., spam) threshold.*

(3) ***Application C*** *may compute a per-ad recommender model [13] based on user-session-based activities, leveraging machine learning libraries over varying windows, on either offline or real-time data.*

Writers of such applications have several unique requirements:

1) Ability to Create Transferable Plans and Logic: Application writers (e.g., application B) need the ability to *transfer* or *operationalize* their offline logic to execute directly over real-time data (or vice versa), with carefully tuned distributed physical plans. They wish to avoid maintaining multiple workflows and systems [25] with an ability to create plans that are transferable, i.e., the plans can be moved from offline to real-time deployments and vice versa.

2) Ability to Execute a Rich Space of Plans: Application writers (e.g., application A) expect high performance with a rich space of plans that exploit intra- and inter-node parallelism, as well as *columnar* [15] execution for performance. Beyond shuffle- or broadcast-based relational plans such as the per-ad count example, one may wish to deploy diverse distributed plans that replicate data in specific ways across machines, such as partial duplication of

*Work performed during internships at Microsoft Research. Ahmed Eldawy is currently at the University of California, Riverside. Abdul Quamar is currently at IBM Research, Almaden.

Proceedings of the VLDB Endowment, Vol. 9, No. 14

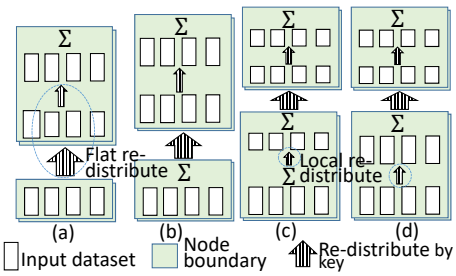

Figure 1: Some distributed plans for grouped aggregation

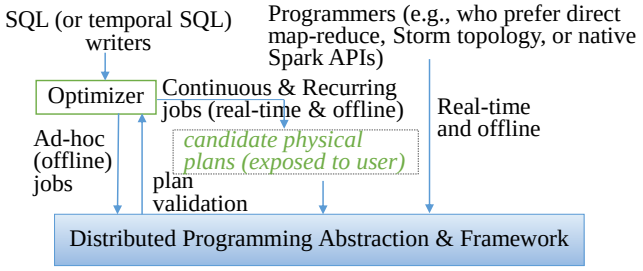

Figure 2: An alternate analytics workflow

data to efficiently handle data-parallel computations such as multi-way joins [17], theta joins [28]), matrix operations, clustering, and neighborhood-centric computations.

3) Ability to Control & Fine-Tune Plans: Since query optimizers do not always produce optimal plans (particularly in the distributed case), commercial DBMSs offer the workaround of *query hinting* [11], where a user can instruct the optimizer to constrain its plan search space. This gives application writers (e.g., application A) some control over their physical plans; this is critical for continuous and recurring queries because these users understand their data and have sufficient lead time to carefully tune the plans before deploying an expensive long-running or recurring job. Other users anticipate specific changes in their workloads, and wish to choose plans, generated either via an optimizer or programmed directly, which are robust to these changes. With only plan hints, the physical plan can be seen (e.g., via *explain plan* [29]), but is not guaranteed to be transferable or easily readable. Further, plan hints only allow imprecise control; hence, some systems such as SQL Server offer a primitive ability to force a specific plan (in XML) [20], but the plans target a single node and are not integrated into the HLL.

4) Ability to Author Complex & Temporal Logic: Beyond relational plans, application writers (e.g., application C) need to deploy plans with richer time-oriented logic, such as windowing by time or in a data-dependent manner (by user session [3]); see §3.7 for an end-to-end example involving online advertising. Further, they often wish to use libraries of operators that implement such non-trivial logic, as well as easily program their own logic (operators), leveraging rich HLL libraries and data-types (e.g., dictionaries of model parameters), in a manner that retains transferability and efficiency.

1.1 Towards an Alternate Workflow

Towards satisfying all these requirements, we argue for a new distributed programming abstraction that unifies two worlds (see Fig. 2): (1) Imperative programmers can code directly against this abstraction for control, while leveraging a rich set of physical operators; (2) SQL query authors can write declarative queries which are translated into a set of candidate distributed physical plans expressed using the same readable abstraction. Users can modify or fine-tune a chosen plan, and possibly validate it against the optimizer (feasible in many cases [20]), as a powerful alternative to query hinting.

Table 1 summarizes today's analytics platforms (§7 has more details). Many customers program explicit dataflows, e.g., via Storm [7] topologies and map-reduce [19] or native Spark [33] programs. While this provides more control, the space of plans is limited by the constrained API (e.g., having to generate keys to simulate joins [10, 17, 28]). Performance is low (e.g., no columnar), making them uncompetitive with DBMSs. Moreover, application-time or SQL support has to be "bolted on" and the burden of writing temporal logic or operators is pushed to the layer above. This lowers performance and complicates transferability. Layers such as Hive [31] compile SQL to the map-reduce API. However, they have poor performance due to the inability to express complex (or columnar) physical plans in the constrained API. Hence, systems such as Impala (or SparkSQL) circumvent map-reduce (or native Spark) and directly process data using specialized DBMS-style plans [18].

1.2 Introducing Quill

Quill (stands for a *quadrillion tuples per day*) is a high-performance distributed platform for streaming tempo-relational analytics, across scales from a single-core to multi-node on the cloud. Experiments (§ 6) on up to 400 cloud machines, and on benchmark and real datasets up to 1TB, show Quill to incur low overheads and outperform SparkSQL [8] by up to orders-of-magnitude for temporal and 6× for relational queries, while supporting significantly expanded temporal querying functionality. Our contributions include:

1.2.1 Sharded Dataset & Compute Abstraction

We propose a sharded temporal dataset and compute abstraction called *ShardedStreamable* (§ 3). This abstraction represents a fixed number of *shards* of the data. Each shard represents a single time-ordered, possibly overlapping, fragment of the dataset, stored as a sequence of columnar batches. The API supports:

- ***Query logic***: A `Query(..)` operation works over one (or two) sharded datasets, accepts a query specification, executes the query independently on every shard (or pair of shards), and produces a new sharded dataset. Queries are specified in the extensible language of Trill (stands for a *trillion tuples per day*), an incremental analytics library that enables relational, progressive (approximate) [14], and temporal [16] processing at best-of-breed performance on one core. Users can easily create new transferable operators, either by providing logic (code) to add and remove tuples to and from state, or by writing an operator that reads and produces a sequence of batches.
- ***Data movement***: A set of cross-shard operations create new datasets with the shard contents organized or duplicated in specific ways. This includes `ReShard(..)` for load-balancing, `Broadcast(..)` for duplicating every shard on every result shard, and `Multicast(..)` for duplicating each tuple on zero or more result shards based on the payload. One can specify physical locations for result datasets to further optimize the physical plans.
- ***Keying***: Each tuple in a shard may optionally be associated with a key. A `ReKey(..)` operation re-keys each shard independently, to associate each tuple with a new key. When a sharded dataset is keyed, a `Query(..)` operation logically executes on shards on a per-key basis. `ReKey(..)` does not move data across shards; a separate `ReDistribute(..)` operator moves data across shards to re-organize shards by key.

By separating incremental query logic specification, a small but rich set of data movement operations, and keying, one can build complex[1] distributed physical plans in a HLL that target different

[1]For compatibility, APIs such as map-combine-reduce-merge and native Spark can be implemented over ShardedStreamable. Iterative computation is supported at the application level like Spark.

Table 1: Sample of today's solutions for distributed analytics

Requirement	Map-reduce	Hive	Trad. DB, Impala	Spark SQL	Native Spark	Data-flow, Flink	Storm	Quill
Rich Temporal Support	No	No	No	No	No	Yes	No	Yes
Incremental	No	No	No	No	Yes	Yes	Yes	Yes
HLL Integration	Yes	No	No	Yes	Yes	Yes	Yes	Yes
Throughput	Low	Low	High	High	Mid	Low	Low	High
Columnar Execution	No	No	Yes	Yes	No	No	No	Yes
Rich Physical Plans	No	No	Some	Some	Some	Some	No	Yes
Programmable Plans	Yes	No	No	No	Yes	Yes	Yes	Yes
Transferable Plans	Dep-ends	No	No	No	Dep-ends	Yes	Yes	Yes

scales, including physical plans of modern scan-based columnar databases, continuous queries (e.g., hybrid symmetric hash and broadcast joins), and distributed settings (e.g., selective data replication for multi-way or theta joins and matrix operations). Critically, Quill's physical plans are *transferable by construction*, between real-time and offline deployments. As a brief preview, the distributed physical plans of Fig. 1(a) and (b) are written in Quill as:

```
(a): s.ReKey(e => e.AdId)
      .ReDistribute().Query(q => q.Count());
(b): s.ReKey(e => e.AdId).Query(q => q.Count())
      .ReDistribute().Query(q => q.Sum());
```

1.2.2 *Layered System Architecture*

Quill uses a loosely-coupled layered architecture (see Fig. 3) that provides a careful separation of responsibilities, *while retaining high performance*, and maximizes component reusability:

Single- and Multi-Core Support: The `Query` operation is an incremental scan of (optionally keyed) batched columnar shards, that can leverage the extensible Trill library (§ 2) for single-threaded temporal logic execution. On top of this functionality, we add a concrete implementation (§4) of ShardedStreamable for multi-core that provides cross-core data movement operations.

Cloud Support: We implement ShardedStreamable for the multi-node cloud setting (§ 5) as a HLL-integrated client library. Unlike traditional systems that use a special master node for coordination, and can become a single point of failure, Quill uses *master-less coordination* of cloud workers using decentralized and replicated resources available in cloud platforms (e.g., tables and queues). This design is a good fit for the pay-as-you-go cloud as it decouples clients, metadata, and workers, and simplifies management. We measure overheads (§6.2) and find this design to be feasible. We leverage the multi-core ShardedStreamable for query execution and data movement. In addition, the client library provides functionality to manipulate clusters using a *virtual cluster* abstraction, and work with shared datasets in memory or on storage.

Component Reuse: This layered architecture enables component reuse: each layer is an independently useful artifact. For example, we earlier reported the Trill library's independent use in diverse environments across Microsoft [15]. ShardedStreamable can similarly be used independently (1) on single- and multi-core, for scaling up or partitioning computation on either a stand-alone machine or a node that is part of an existing distributed dataflow system; and (2) as part of the Quill multi-node cloud platform described in §5.

While the abstractions we propose in this paper are agnostic to real-time vs. offline, our current implementation of the Quill cloud platform is optimized for offline logs and, like Spark, recovers from failure via lineage tracking and re-execution (§5.4).

2. BACKGROUND: THE TRILL LIBRARY

Quill's inner layer (Fig. 3) leverages a library responsible for single-threaded processing. While Quill is in principle agnostic to the library, we nail down the data format by reusing and extending Trill's data model, and add new operators for keying and cross-shard data movement. This section summarizes the unmodified design of Trill; our enhancements to it for Quill are covered in Section 4. Trill is written in the high-level-language (HLL) of C#, and thus supports HLL data-types and logic. By default, libraries do not own threads; they perform computation on the thread that feeds data to them.

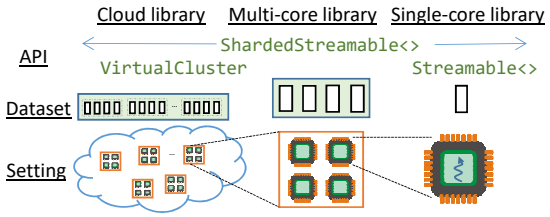

Figure 3: Quill overview

2.1 The Trill Data Model

A source of data with payload type `TP` is represented as an instance of a class called `Streamable<TP>`. Continuing our advertising example, we may use a C# payload type for click logs:

```
struct AdInfo { long Time;  long UserId;  long AdId; }
```

This data source is of type `Streamable<AdInfo>`. Physically, a stream consists of a sequence of *columnar batches*. A batch holds a set of columns (arrays) to hold a timestamp and window description (as a time interval), a pre-computed *grouping key* of type `TK`, a 32-bit hash of the key, and each payload field as an individual column (we generate the batch class using dynamic code generation). The key for ungrouped streams is a special empty struct called `Empty` with a hash of 0. An *absentee bitvector* identifies inactive rows in the batch. For example, the generated batch for `AdInfo` looks like:

```
class ColumnarBatchForAdInfo<TK> {
 long[] SyncTime; long[] OtherTime; // timestamp & window
 TK[] Key;          int[] Hash;       // key and hash
 long[] BitVector;                   // bitvector
 long[] Time; long[] UserId; long[] AdId; // payload
}
```

Trill uses memory pools to recycle and share arrays between operators. For example, when we receive rows of type `AdInfo`, we allocate three long arrays (Time, UserId, and AdId) from the pool. During data ingestion, the user can specify that the Time field represents our application time, using a lambda expression [26] `e => e.Time`; an *anonymous function* to compute application time from the payload. The expression tree is available at query compile-time, so we can recognize that SyncTime can point to the same array as Time, with an added reference count. Operators return arrays to the pool when done, and use copy-on-write to update shared arrays.

2.2 The Extensible Trill-LINQ Language

Trill's language is Trill-LINQ [15], which is exposed as methods on an instance of `Streamable<TP>`. Each method represents a physical operator (e.g., `Where` for filtering) and returns a new `Streamable` instance, allowing users to chain a physical plan. For instance, with a data source `s0` of type `Streamable<AdInfo>`, we can filter a 5% sample of users using the `Where` operator:

```
var s1 = s0.Where(e => e.UserId % 100 < 5);
```

The lambda expression in parentheses is from the type `AdInfo` to a boolean value specifying for each row (event) `e` in the stream that it is to be kept in the output stream, `s1`, if `e.UserId % 100 < 5`.

An operator accepts and produces a sequence of columnar batches. An operator is dynamically generated C# code that inlines lambdas (such as the `Where` predicate) in tight per-batch loops to operate

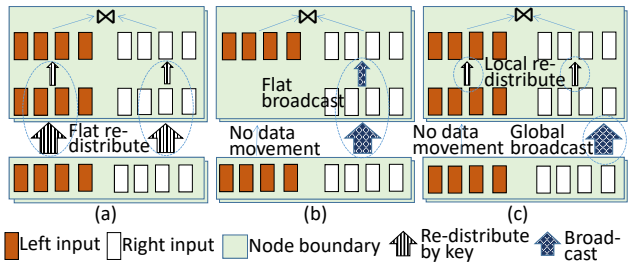

Figure 4: Some distributed plans for join

directly over columns for high performance. Trill provides a rich set of built-in relational operators (e.g., join) as well as new temporal operators for defining windows and sessions. Trill-LINQ is extensible in several ways. First, users can express user-defined aggregation logic by providing lambdas for accumulating and de-accumulating events to and from state. Such logic is executed over columnar batches using an automatically generated *snapshot* operator that maintains per-group state and inlines these lambdas in a tight loop [15]. Second, advanced users can write new operators that accept and produce a sequence of (grouped) columnar batches. Note that Trill operators understand grouping; e.g., a `Count` operator (also implemented using our user-defined snapshot framework) outputs a batched stream of per-key counts. Further, every operator is transferable between real-time and offline by construction.

Trill supports a `GroupApply` operation that executes a grouped sub-query (GSQ) on each sub-stream corresponding to a distinct grouping key. For example, we can compute a per-ad count as:

```
var s2 = s1.GroupApply(e => e.AdId, q => q.Count());
```

Here, the first lambda specifies the grouping key (AdId) and the second lambda specifies the GSQ. Unmodified Trill exposes grouped computation to query writers only within the context of a GSQ. `GroupApply` is implemented internally by preceding the GSQ by an operator that updates batches to have the user-specified grouping key (nested with the previous grouping key), and following the GSQ with an operator that un-nests the grouping key.

3. THE SHARDED STREAMABLE API

We wish to create a HLL-integrated abstraction that enables high-performance columnar execution, leverages the extensible language of Trill-LINQ, and supports the creation of a rich set of distributed physical plans that are transferable between real-time and offline. In Section 1, we illustrated some physical plans for grouped aggregation. For multi-input operations, the space of distribution strategies is even richer. For instance, Fig. 4 considers some physical plans for the join operation. Fig. 4(a) is a distributed symmetric hash join, Fig. 4(b) broadcasts the smaller side globally, while Fig. 4(c) uses a hybrid strategy of broadcast across machines and hashing within a machine. The choice of plan itself is orthogonal and may be based, for example, on intra- and inter-node data transfer bandwidths.

3.1 The ShardedStreamable Abstraction

As introduced in Section 1.2.1, the immutable ShardedStreamable abstraction represents a distributed dataset as a set of shards. Each tuple in a shard is a payload of type `TP`, and may *optionally* be associated with a key of type `TK`. Even when payloads are associated with keys, sharding is orthogonal: there is no constraint on which shards may contain which keys. Physically, a shard is organized as a sequence of columnar batches, as described in Section 2.1.

As a warm up, consider the case where tuples are not associated with any key. Such a ShardedStreamble, for payload type `TP`, is of type `ShardedStreamable<TP>`. We may initially create a ShardedStreamable in several ways, e.g., by pointing to a directory in storage. In our running example, we can construct a `ShardedStreamable<AdInfo>` named `ss0` using a HDFS path to the dataset as follows:

```
var ss0 = createDataset<AdInfo>("/data/hdfs/adinfo");
```

We classify operations over ShardedStreamable as *transformations* and *actions*, similar to Spark [33]; see Table 2. Transformations do not perform computation, but return new ShardedStreamble instances of the appropriate type to allow type-safe composition. For example, `createDataset` is a transformation because it does not actually load the data in main memory, it simply associates the specified path to a ShardedStreamable instance. Actions, on the other hand, trigger the immediate computation of all the transformations issued until that point, and block until the computation is done. The result is a new dataset, since all datasets are immutable. For now, we assume that all operations return new datasets with the same number of shards as the input (we revisit this in Section 3.5).

3.2 Basic Transformations

3.2.1 *Query (over a single input)*

The `Query` transformation on a ShardedStreamble accepts a lambda expression that represents an *unmodified* query in the single-core engine's extensible language (e.g., Trill-LINQ) that we wish to execute *independently* on every shard. When triggered, the query executes over each shard independently in columnar fashion, and there is no data movement across shards. For example, suppose we wish to select a 5% sample of users, and select only two payload columns (UserId and AdId) into a new type `UserAd`:

```
var ss1 = ss0.Query(q => q.Where(e => e.UserId % 100 < 5)
      .Select(e => new UserAd { e.UserId, e.AdId }));
```

Here, `ss1` is of a new type `ShardedStreamable<UserAd>`. It is important to note that all transformations are strongly typed and fully type-safe; query writers get auto-completion and type-checking support during query authoring, and are prevented from making common mistakes during query authoring. They are free to seamlessly use HLL libraries and methods in all operations as well.

3.2.2 *Query (over multiple inputs)*

We support multi-input queries by exposing a two-input version of the `Query` transformation, which operates over two ShardedStreamable instances, and accepts a two-input lambda expression as parameter that represents an unmodified two-input query (e.g., join) in the single-core engine's language, that we wish to execute over pairs of shards from the two input datasets (compile-time properties enforce inputs to have the same number of shards). The operation produces a single sharded dataset as output. We provide an example of this operation in the context of `Broadcast`, described next.

3.2.3 *ReShard, Multicast, Broadcast*

We now present cross-shard data movement operations that create new datasets with the shard contents organized or duplicated in specific ways. `ReShard` does a blind round-robin "spray" of every input shard's content across a set of result shards, and is used to spread data evenly across shards. `Multicast` is a powerful operation that sends each tuple in each input shard to *zero or more* result shards, based on a user-provided lambda expression over the payload. Using `Multicast`, one can selectively replicate tuples to result shards to implement theta-joins [28], parallel multi-way joins [17], and matrix operations. Finally, `Broadcast` sends all the data in each input shard to *every* result shard. All data movement operations retain the timestamp order of shards during transformations. For example, suppose we have a sharded dataset `ss1` of payload `UserAd`, that

we wish to join to a reference dataset `rr0`, of type `AdData`, that contains per-AdId information such as bids and keywords. If the reference dataset is small, we can keep the larger dataset `ss1` in place and execute a *broadcast join* by broadcasting `AdData` to all shards, followed by the equi-join as a two-input `Query`, as follows:

```
var rr1 = rr0.Broadcast();
var ss2 = ss1.Query(rr1,
   (left, right) => left.Join(right, e => e.AdId, ...));
```

Here, we use the two-input Trill-LINQ Join operator that takes the equi-join key (`e => e.AdId`) as a lambda parameter.

3.3 Key-Based Transformations

Abstractions such as map-reduce expose an explicit key per tuple in order to enable partitioned execution. For logical queries, keying the data also allows us to execute queries in a grouped manner. To enable both functions, we support keys as a first-class citizen in ShardedStreamable. Each tuple may be associated with a key of type `TK`, and such a dataset is represented by type `ShardedStreamable<TK, TP>`, where `TK` is the key type and `TP` is the payload type. Every row in the dataset has a payload of type `TP`, and a key of type `TK`. This simply means that the `Query` operation on a shard is aware of keying and logically executes as a group-by-key query[2]. The output is a dataset with keys unchanged. Both the key and its hash value for every tuple are computed and materialized as columns in the dataset. We separate the notions of keying the data (`ReKey`) and re-distributing the data across shards based on key (`ReDistribute`):

3.3.1 *ReKey*

The `ReKey` transformation accepts a lambda to select a new key of type `TK2` from the payload, and creates a `ShardedStreamable<TK2, TP>` with the new key. When executed, ReKey does not move data across shards; rather, it just modifies each tuple in the result shards to have a different key. ReKey is very efficient as it operates independently per shard and often only involves a per-key hash pre-computation (Section 4 has details). In our example, we may ReKey the dataset `ss1` by AdId as follows:

```
var ss3 = ss1.ReKey(e => e.AdId);
```

3.3.2 *ReDistribute*

`ReDistribute` reorganizes data across shards, and outputs a dataset with the same key and payload type as its input. When executed, `ReDistribute` re-distributes data across shards so that all rows with the same key reside in the same shard. By default, `ReDistribute` uses hash partitioning on the key (using hash values computed during `ReKey`) to move the data. Re-distributed datasets have the property that different shards contain non-overlapping keys.

In our running example, since we have already re-keyed the dataset by AdId (`ss3`), we can re-distribute it across the shards by AdId, and then compute a per-AdId count, as follows:

```
var ss4 = ss3.ReDistribute();
var ss5 = ss4.Query(q => q.Count());
```

As a minor variation, we could perform local-global aggregation trivially with this API:

```
var ss6 = ss3.Query(q => q.Count()).ReDistribute()
              .Query(q => q.Sum());
```

Here, we first compute per-key counts independently in every local shard, and then re-distribute the counts across shards by AdId, before computing the global per-AdId sums. Apart from ease of specification, this example hints at how the separation of `ReKey` and `ReDistribute` can provide efficiency (see Section 4 for implementation details): the key and hash are computed and materialized exactly once (at `ReKey`), and are used to: (1) compute the local per-key count; (2) partition data by hash across shards; and (3) compute the global per-key sum in bulk. In contrast, with map-reduce: (1) `Map` emits fine-grained ⟨key, 1⟩ pairs; (2) `Combine` computes key hashes and builds a hash-table (by key) of raw value lists, to periodically aggregate local per-key counts; (3) the shuffle re-hashes keys to partition the data to reducers; and (4) on the reduce side, data is re-grouped by key, using either a sort or hash, before invoking `Reduce` repeatedly (per-key) to compute counts. Performance is limited because of expensive fine-grained intermediate data creation, hash computation, and per-row method invocation.

[2]For efficiency, operators in the library implementing `Query` need to be aware of grouping; we cover implementation details in Section 4

Table 2: ShardedStreamable operations

	Operation	Description
Transformations	`Query`	Applies an unmodified query over each keyed shard.
	`ReShard`	Round-robin movement of shard contents to achieve equal shard sizes.
	`Broadcast`	Duplicate each shard's content on all the shards.
	`Multicast`	Move tuples from each shard to zero or more specific result shards.
	`ReKey`	Changes the key/hash associated with each row in each shard.
	`ReDistribute`	Moves data across shards so that same key resides in same shard.
Actions	`ToMemory`	Materialize transformation results into main memory.
	`ToStorage`	Materialize results to specified path.
	`ToBinaryStream`	Materialize results to IO-streams.
	`Subscribe`	Materialize and apply the provided lambda expression to each result row.

As another example, suppose we wish to join (on AdId) our original AdInfo dataset `ss0` to the reference dataset `rr0`, but wish to use the familiar distributed hash-join. We re-key both sides to AdId, re-distribute both sides, and execute the join as follows:

```
var ss7 = ss0.ReKey(e => e.AdId).ReDistribute();
var rr2 = rr0.ReKey(e => e.AdId).ReDistribute();
var ss8 = ss7.Query(rr2, (l, r) => l.Join(r));
```

The first two lines re-key and re-distribute the input datasets by AdId. Next, the `Query` transformation runs over `ss7` and `rr2`, and applies a join query in Trill-LINQ (represented by the two-input lambda expression) to produce a new sharded dataset `ss8`.

3.4 Actions

Actions are used to materialize query results. `ToMemory` stores the result of the computation in a (potentially distributed) in-memory dataset, whereas `ToStorage` stores the result in a persistent store. Both are useful for sharing datasets and transferring to other workflows. `ToBinaryStream` takes an array of IO-streams (a standard abstraction for input and output devices such as network or disk) as parameter and outputs each shard in an efficient binary format to the corresponding IO-stream in the array. `Subscribe` accepts a lambda expression that is invoked for every tuple in the materialized result, and is useful for clients to operate directly on results as part of their application (see [12] for examples of actions).

3.5 Location-Aware Data Movement

We have until now assumed that data movement occurs from and to the same set of fixed shard locations. We relax this organization by letting data movement operations accept an optional *location descriptor* argument that identifies where the data moves to. For example, in multi-core, we may re-distribute 16 input shards to 4 output shards to utilize one socket. In multi-node, we may re-distribute a reduced dataset to a new *virtual cluster* with fewer machines. Further, locations can be optionally viewed as a layered

organization: a set of *global* shards, each of which consists of a set of *local* shards. Thus, all data movement operations take an optional "scope" argument, which can be either *local* or *global*. If unspecified, data movement assumes a *flat* movement across all shards.

This layering allows us to express complex inter- and intra-node data movement. Fig. 1(a) and (b) were covered in Section 1. Fig. 1(c) aggregates per-core, re-distributes across shards within a machine and re-aggregates before a global re-distribute for the final aggregation. This strategy makes sense if keys are duplicated across the shards within a machine. Fig. 1(d) is similar, except it does not perform per-core aggregation, which may be superior for a large number of groups, where the memory cost of building a large hash table per core (if we aggregated before re-distributing by key) would be high. These plans are expressed over the keyed dataset `ss3` as:

```
(c): ss3.Query(q => q.Count()).ReDistribute(Local)
       .Query(q => q.Sum()).ReDistribute()
       .Query(q => q.Sum());
(d): ss3.ReDistribute(Local).Query(q => q.Count())
       .ReDistribute().Query(q => q.Sum());
```

Other examples, such as the joins in Fig. 4, are covered in [12].

3.6 Revisiting Transferability

We illustrate how the Quill API provides plan transferability. Users read from a real-time data source instead of offline files or caches, by using a variant of the `loadDataset` call that takes a real-time stream constructor as parameter:

```
createDataset<AdInfo>(p => new KafkaToShardedStr(p));
```

This example reads real-time data from Kafka [6] (a messaging service). The lambda takes a partition id as argument and constructs a ShardedStreamable that is capable of delivering a sequence of columnar batches from that partition. All ShardedStreamable operations are incremental by construction, including `Query` because it leverages Trill's physical operators and extensibility framework which are transferable by construction [15]. Thus, a user's offline Quill logic can work over real-time and vice versa.

3.7 End-to-End Complex Temporal Query

As a more complex temporal query, consider the problem of sanitizing our advertising dataset `ss0` (bounded or unbounded) to eliminate *bots*, and computing a per-ad count of sanitized users over each Z second period. A bot is a spurious (often automated) user who has clicked on more than X ads in a short timeframe (say, last Y secs). We first `ReKey` and `ReDistribute` to perform bot detection on a per-user basis (this example ignores optimizations discussed earlier, such as pre-aggregation, for simplicity), and express the temporal logic using Query. We then use the temporal `WhereNotExists` [15] Trill operator with Quill's two-input `Query`, to produce a sanitized input log. Finally, we `ReKey` and `ReDistribute` by AdId to compute our desired hopping window result. The entire query plan executes in a pipelined manner because we do not have an action until the final `ToStorage`.

```
var w0 = ss0.ReKey(e => e.UserId).ReDistribute();
var w1 = w0.Query(s => s.SlidingWindow(Y)
                  .Count().Where(c => c > X));
var w2 = w0.Query(w1, (l, r) => l.WhereNotExists(y));
var w3 = w2.ReKey(e => e.AdId).ReDistribute()
           .Query(s => s.HoppingWindow(Z).Count())
           .ToStorage(...);
```

4. SINGLE-NODE ARCHITECTURE

In this section, we describe how ShardedStreamable is architected by leveraging and extending the Trill query library. We first describe how we extend the `Streamable<TP>` class exposed by Trill with keying and new runtime operators, and then cover our implementation of ShardedStreamable operations using these operators.

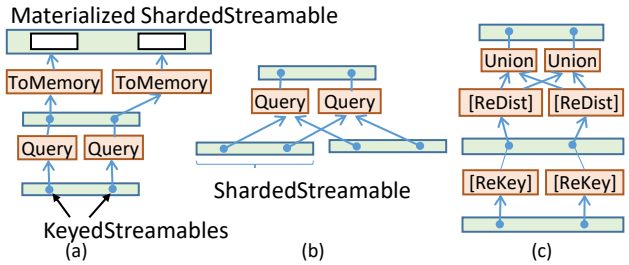

Figure 5: Constructing single-node physical plans

4.1 From Streamable to KeyedStreamable

We represent a single shard of a dataset by an instance of type `KeyedStreamable<TK, TP>`, where `TK` is the key type and `TP` is the payload type. An unkeyed shard uses the special key of `Empty` (cf. Section 2.1). Thus, `ShardedStreamable<TK, TP>` on a single machine is simply an array of KeyedStreamable instances.

KeyedStreamable is an unpartitioned dataset with grouping key set to `TK`, and requires a concrete implementation (i.e., query engine) to support the underlying query language. *KeyedStreamable* is in principle agnostic to the actual query engine that implements this abstraction. We extended Trill's notion of `Streamable<TP>` to implement KeyedStreamable. A query on `KeyedStreamable<TK, TP>` receives batches of keyed tuples, and logically executes as a grouped query, on a single thread. This modification was trivial because the concept of grouped computation already exists internally in the Trill library (e.g., inside a `GroupApply`), as described earlier.

Actions such as `ToMemory` already exist in `Streamable<>`. However, several new physical operators, denoted with `[...]`, are needed in `KeyedStreamable<TK, TP1>` to support ShardedStreamable:

- `[ReKey]` takes a grouping key selector as parameter and produces a new KeyedStreamable with the updated grouping key. For example, when `[ReKey]` with key selector `e => e.AdId` is applied on the Trill stream `s0` of type `Streamable<AdInfo>`, the result is a stream of type `KeyedStreamable<long, AdInfo>`, where `long` is the type of the AdId (key) field. `[ReKey]` first executes a few constant-time operations (per input batch) in this example: setting the Key array to point to the AdId array, and setting all other arrays to point to corresponding input arrays. This is followed by the computation of the Hash array in the result batch.
- `[ReDistribute]` accepts a count M as parameter, and outputs an array of M KeyedStreamable instances, one per destination shard. `[Multicast]` operates similarly, but leverages the user-provided lambda to determine destination shards for each tuple. `[ReShard]` and `[Broadcast]` similarly output an array of M KeyedStreamable instances. Their implementation details over columnar batches are covered in our technical report [12].

4.2 Physical Plan Construction and Execution

Consider a ShardedStreamable with N shards. `Query` takes the query specification as a lambda from `KeyedStreamable<TK, TP1>` to `KeyedStreamable<TK, TP2>`, applies the query lambda on each of the N KeyedStreamable instances, and packages the results into a new ShardedStreamable instance, as shown in Fig. 5(a). `ReKey` is applied similarly. The two-input `Query` operation works similarly, but it constructs N *pairs* of KeyedStreamable instances, which are combined using the two-input query lambda, and finally packaged into a new ShardedStreamable instance, as shown in Fig. 5(b).

`ReDistribute`, `ReShard`, `Multicast`, and `Broadcast` take a location descriptor that represents the number of new shards, say M (the default keeps the sharding unchanged, e.g., $M = N$). We first invoke the corresponding operations on each KeyedStreamable,

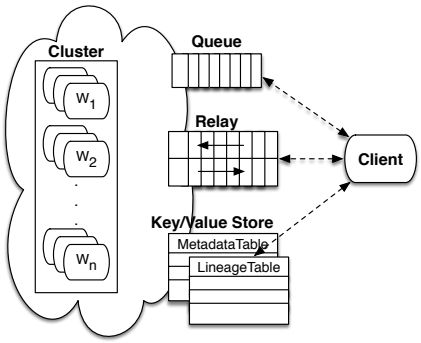

Figure 6: Overview of Quill's cloud architecture

specifying M as location, resulting in N arrays, each of which is an array of M KeyedStreamable instances. We then use *temporal union* operator to merge these into M shards in timestamp order; these shards are packaged into the result ShardedStreamable. Temporal union optimizes for the case where timestamps across streams overlap only across batch boundaries, and can perform the union by simply swinging pointers to batches, instead of a memory copy of the contents. Fig. 5(c) shows a shuffle operation, that consists of re-key followed by re-distribute and union. Finally, actions are executed using the corresponding methods on KeyedStreamable [12].

5. QUILL FOR THE CLOUD

The next layer of Quill is our distributed platform that targets execution on a public pay-as-you-go cloud vendor (we use Microsoft Azure [27]). Quill exposes a HLL-integrated client library to users to manage and query sharded datasets in the cloud. The library communicates with cloud worker instances to provide a seamless programmatic querying functionality to users.

5.1 System Overview

We designed Quill for the cloud as a fully decentralized system that communicates directly with the client. Unlike traditional big data systems that use a special master node for coordination (and can become a single point of failure), all control flow communication happens through decentralized and replicated resources in Quill. In particular, we leverage: (1) *tables* (that implement a key-value store); (2) *queues* (that expose a FIFO message abstraction); and (3) *relays* (that implement the publish/subscribe paradigm of topic-based message broadcast). These distributed and replicated resources are commonplace in all major cloud vendors today, and incur very little overhead, particularly since they are used only in the less performance-sensitive control flow paths.

Our overall design is shown in Fig. 6. There are two major entities in the picture: a client that runs the *client library* and N workers that execute the *cloud backend library*. The client library supports the ability to create clusters, create and share datasets, and implements the ShardedStreamable abstraction as well. It implements components necessary to communicate with the workers in the cloud via the decentralized resources, receive feedback on progress and errors, and communicate these back to the user. Workers use a cloud backend library that communicates with the client via the same resources, handles inter-node network communication via TCP, and enables query processing using multi-core ShardedStreamable.

5.2 The Client Library

The functions exposed by the client library can be divided into different groups: (i) those that are used to manipulate clusters, i.e. provision, scale up/down, and decommission clusters on-demand;

```
// Application 1
var cluster1 = Quill.createCluster(new Nodes[10] { .. });
var adInfo = Quill.createDataset<AdInfo>
             ("/data/hdfs/adinfo", cluster1).ToMemory();
var counts = adInfo.ReKey(e => e.AdId).ReDistribute().
     Query(e => e.Count());
var cluster2 = Quill.createCluster(new Nodes[2] { .. });
counts.ReDistribute(cluster2).ToMemory("adcounts");
Quill.removeDataset(adInfo);
Quill.tearDown(cluster1);

// Application 2
var counts = Quill.searchDataset("adcounts");
counts.Subscribe(e => Console.WriteLine(e));
```

Figure 7: Two hypothetical applications using Quill

(ii) functions used to work with datasets, such as search, create or delete; and (iii) the ShardedStreamable API. We describe these operations using two hypothetical applications, shown in Fig. 7.

Virtual Clusters. The first step for an application or user is to create and provision a cluster of cloud machines (called *workers*) for analytics. We organize workers into *virtual clusters* (VCs) that represent groups of machines that operate as a single unit. Each VC has a name, and is associated with a broadcast topic in the pub/sub relay. All workers in the virtual cluster are subscribed to the topic. The client library exposes functions to create and tear-down VCs on demand (see Table 3). Re-sizing is done by creating a new VC, moving datasets if needed using `ReDistribute` or `ReShard`, and tearing down the old VC. Information on VCs is stored in a cloud table called the *MetadataTable*. In our example, application 1 creates a VC with 10 machines. Virtual clusters can also serve as a location descriptor for data movement operations. For example, application 1, after the analytics, re-distributes the result to a smaller VC with 2 machines and tears down the larger VC.

Datasets. We support two types of datasets, both of which implement the ShardedStreamable API. A disk dataset (`DiskDataset`) points to storage, whereas an in-memory dataset (`InMemDataset`) represents a dataset that is persisted in distributed main memory. Metadata about these datasets is stored in the MetadataTable. Clients can use several available functions to load the required datasets. For example, it is possible to search the metadata for available shared datasets, add new datasets that can be analyzed later or left for other users that may be interested. We also allow users to delete old datasets and reclaim memory. For example, application 1 creates a `DiskDataset`, loads it into the 10 machine VC as an `InMemDataset`, and subsequently deletes it. The result dataset, named "adcounts", is later retrieved and used by application 2.

ShardedStreamable. Users write analysis queries using the *ShardedStreamable* API. Query execution is triggered when a user writes an *action* such as `ToMemory` or `ToStorage`, which creates an instance of an `InMemDataset` or `DiskDataset` respectively. `Subscribe` can be used to bring the raw results back to the client (cf. Section 3.4). For example, application 1 writes "adcounts" to memory, and application 2 retrieves results to display on the client console.

5.3 Implementing ShardedStreamable

When an action is triggered, Quill extracts the dataflow graph (DAG) of transformations expressed by the query, and groups them into *tasks* to construct the distributed physical plan. A task in Quill is a unit of work that is sent from the client to the workers via the relay, and conveys an intra-node physical sub-plan (again, expressed using ShardedStreamable). Transformations are pipelined and sent as a single task. A data movement operation such as `ReDistribute` breaks the pipeline into two special tasks: one for the sender and another for the receiver (which could be in a different VC).

Table 3: Client library API

	Operation	Description
Cluster	`createCluster`	Creates a new virtual cluster with a given number of machines.
	`tearDown`	Tears down the cluster.
Dataset	`searchDataset`	Searches dataset by id or name.
	`createDataset`	Allows to create new datasets from local or remote paths.
	`removeDataset`	Deletes an existing dataset.

The transformations are serialized and published to the broadcast relay under the appropriate topic. Workers that are subscribed to the same topic receive the tasks and execute them. Workers maintain TCP connections and use `ToBinaryStream` and `FromBinaryStream` for efficient columnar compression and serialization (see [12] for more details). On finishing, they write to a table indicating their id. With this mechanism, Quill's client knows that the query is finished when the table contains the ids of all the workers in the cluster.

5.4 Fault Tolerance and Optimization

Fault tolerance. Quill's model of fault tolerance is as follows: the client library is completely decoupled from the workers, and submits tasks atomically via cloud structures. The client also logs all query transformations in a decentralized table called the *LineageTable*. Jobs run on workers, and a client can disconnect or fail without affecting job execution. Workers register heartbeats periodically with the MetadataTable; when the client (or another worker) finds a node without a heartbeat, it is marked as dead, a replacement node is allocated, and the lineage information is used to recompute datasets on the node. The MetadataTable also needs to be updated to reflect the new locations for restored dataset shards. Handling fault-tolerance for real-time queries is an area for future work; here, we already support checkpointing primitives in Trill, and these primitives could, for example, be coordinated with a replay mechanism [1] to recover from failures of such queries.

Optimization. Quill's plans are physical; the functional Sharded-Streamable invocations result in a tree of operators as an expression tree in the high-level language. This means we can build layers to (a) programmatically translate SQL or other high-level languages into this representation; and (b) apply visitor-based expression tree transformations to implement planner rules that preserve semantics. This process could be based on a cost model, using which the visitor may push down predicates, reorder operators, and select the number of shards. Systems such as Apache Calcite [5] (or more generally, optimization frameworks such as Cascades [22]) may be adopted for our plans as well, to implement the workflow of Fig. 2.

Other Discussion. We cover efficient data loading, minimizing overhead, and handling query errors in our technical report [12].

6. EVALUATION

Our goal is to (a) evaluate the overheads with the master-less design (§6.2); (b) understand the performance of relational-style (§6.3) as well as temporal (§6.4) analytics queries on large bounded datasets, in comparison to a state-of-the-art big data analytics platform; (c) understand Quill's shuffle and data loading performance (§6.5). We use Spark v1.4 with the SparkSQL [8] interface as our baseline, because it provides high-performance columnar execution, and was recently shown [8] to outperform other big data systems, including native Spark [33], Impala [23], and Shark [32] (earlier work [33] also showed native Spark to outperform map-reduce by up to 10×). Note that Quill supports programming the distributed plan unlike SparkSQL, where the optimizer's plan cannot be modified. Except where indicated, we use SparkSQL's *explain* command to get its physical plan and use the same plan in Quill.

6.1 Setup and Workloads

We implemented Quill to target Microsoft Azure [27]. We run experiments in cloud-provisioned clusters of up to 400 D1 nodes or 40 D14 nodes located in the West US region. Each D14 instance has a 16-core Intel Xeon E5-2660 CPU and 112 GB RAM, while D1 instances (used for overhead experiments) have 1 core. All the nodes have 10 Gbps NIC bandwidth. Unless otherwise mentioned, we run the Quill client on a remote (non-cloud) machine in North-West US. However, the Spark client (command line) is co-located in the West US datacenter. Datasets are stored in Azure storage, but are pre-loaded into main memory before the experiments.

We tuned Spark to use in-memory compression and columnar representation, which significantly improved the performance for the queries presented in this section. Both systems use the same number of workers; we additionally made sure to configure the Spark master node with enough memory so that this is not a limiting factor. For both Spark and Quill, we tune the garbage collector to reduce its impact on performance. It is worth noting that the performance results we show for Spark (we highly optimized it for performance) are several factors higher than the numbers reported in the literature [8] for a previous version of the system. Finally, we repeat experiments 10 times, and show the average with error bars.

SQL big data benchmark. We use the big data benchmark proposed by Pavlo et al. [30]; this benchmark contains typical analytical SQL queries and is also used in the SparkSQL paper [8]. This dataset consists of a *rankings* and *uservisits* table, with the schemas shown below. We implemented the benchmark in both Quill and Spark and ran the queries as specified—maintaining a fixed dataset size per node. To show the scalability of both systems, we increase the cluster size until we reach dataset sizes of 1 TB.

```
struct UserVisits {
   String sourceIP;     String destURL;      long date;
   int duration;        float adRevenue;
   String userAgent;    String countryCode;
   String languageCode; String searchWord;  }
struct Rankings {
   String pageURL;  int pageRank;  int avgDuration; }
```

We evaluate the performance of the different systems with the *scan*, *aggregate* and *join* queries presented in the original benchmark, varying different parameters as described later.

GitHub events dataset. We use the GitHub archive [21] to evaluate temporal queries. This 0.5 TB dataset contains 25 types of events registered by GitHub since 2011. Each event correspond to a user action, such as create repository, push commits or watch events (see [12] for more dataset details).

6.2 Masterless Design Evaluation

In this section, we evaluate the overhead of using decentralized cloud structures for coordination in the masterless Quill design.

1) Latency of Decentralized Scheduling. On receiving a new query (workflow), Quill uses a broadcast relay to distribute the operation to all the workers, which execute components of the workflow. On completion, each worker adds an entry to the Azure table (MetadataTable), which the client checks to report query completion. We evaluate the overhead of this round-trip by issuing an empty task that immediately reports back. In Figure 8, we vary the number of Azure D14 instances from 10 to 40, and report average latency with error bars. We report latencies when the client is outside vs. inside the datacenter. As expected, larger clusters incur more overhead, but the average latency is less than 300ms (inside datacenter) even with 40 nodes, which is small compared to the expected completion time of non-trivial analytical tasks (§6.3). For comparison, we experimented with using a service bus queue instead of an Azure table

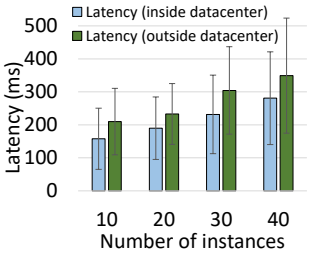

Figure 8: Scheduling overhead; target cluster sizes.

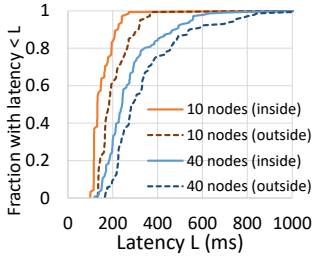

Figure 9: Scheduling overhead; inside vs. outside datacenter.

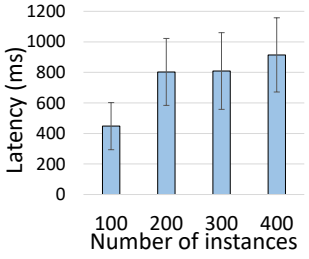

Figure 10: Scheduling overhead; larger cluster sizes.

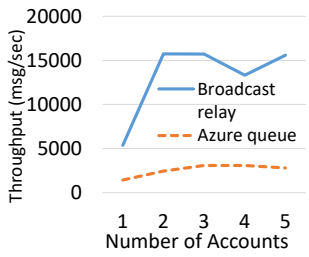

Figure 11: Throughput of cloud structures.

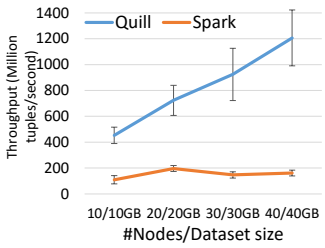

Figure 12: Scan query with a selectivity of 5%

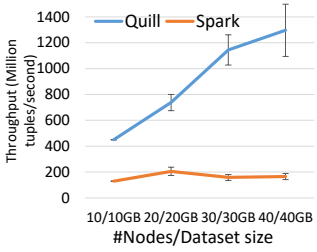

Figure 13: Scan query with a selectivity of 50%

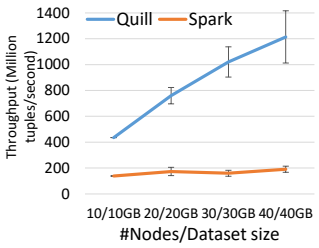

Figure 14: Scan query with a selectivity of 95%

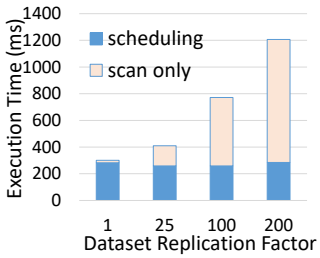

Figure 15: Scan latency split-up, varying dataset size

for reporting completion as well, and found the average latency for 10 instances to be 340ms within the datacenter (slightly higher than using tables). Further, we observe that running the client outside the datacenter increases latencies by around 70 ms, due to the additional latency incurred by the distance.

2) Effect of Client Location. Figure 8 includes latencies when the client runs outside the datacenter; we see that average latencies are around 70ms higher, but are less than 350ms even for 40 nodes. We next show the detailed overhead effect when running the user client outside vs. inside the datacenter. We use a cluster size of 10 and 40 machines, and report the CDF of latencies for inside and outside datacenter in Fig. 9. For instance, we note that with 10 (40 resp.) nodes, 85% of latencies are less than 200ms (400ms resp.) when the client is inside the datacenter.

3) Latency with Larger Cluster Sizes. While we target clusters with less than 100 machines, we next verify scalability for even larger deployments of up to 400 workers. Given a limit on the total number of cores available to us, we use Azure D1 instances with 1 core, and vary the number of workers from 100 to 400. Figure 10 shows the results from inside the datacenter. We see that latency overhead is less than 1 second even with 400 workers, which would provide 44TB of memory if we used D14 instances.

4) Throughput of Decentralized Structures. We next measure the throughput, in terms of number of messages sent and received per second, using our cloud structures (Azure queues and broadcast relay). We use a single client machine with multiple threads to send and receives messages up to 64 bytes long. Figure 11 shows throughput as we increase the number of structures to get more parallelism). We see that even with one structure, we achieve more than 1000 messages per second, which is sufficient for our infrequent coarse-grained control flow messages.

Cloud structures have low latency overhead and high throughput, and can serve as a masterless control flow mechanism.

6.3 Relational Analytical Queries

To understand the performance and scalability of Quill, we run the three queries defined in the SQL Big Data Benchmark changing the selectivity in each case. The dataset size is fixed on a per node basis to conform the benchmark description. This means that we keep around 25 GB (152M rows) of the *uservisits* data and 1 GB (18M rows) of the *rankings* data per node. With this configuration we run the queries in 4 different cluster sizes of 10, 20, 30 and 40 nodes, which translates into datasets of 250, 500, 750 and 1000 GB respectively. We also run queries with larger data sizes that, although not specified in the benchmark, helps to understand better the performance characteristics of the systems.

6.3.1 *Scan Queries*

1) Scan performance. The first query defined by the benchmark is the scan shown below, with selectivities of 5%, 50% and 95%.

```
SELECT pageURL, pageRank FROM Rankings
WHERE pageRank > X;
```

Both systems are columnar and scan only the query predicate column – around 72MB of pageRank data per machine (18M rows × 4 bytes per row). Fig. 12,13, and 14 show the results for Quill and Spark. The throughput in both systems is limited by overheads.

In Quill, with 10 nodes, over 93% of the time is spent on scheduling; the total time using a client outside the datacenter is 300ms, while the actual scan takes just 19ms. The overhead is slightly higher than for an empty query due to additional costs such as client plan construction, task and metadata serialization, and lineage tracking. The bandwidth of the scan alone (ignoring scheduling overheads) is also low (around 3.8GB/sec) because of the small dataset size. Since scheduling overheads in Quill increase slowly with number of nodes (see §6.2), scan throughput increases as the cluster grows. In case of Spark, the overhead is higher, and grows faster with cluster size, preventing its scalability (at 40 nodes, Quill is $\sim 6\times$ faster).

2) Scans on larger data. To evaluate performance with non-trivial compute, we ran additional scan queries with a bigger dataset per node. We first micro-benchmark Quill by varying the replication factor R_F of the rankings dataset (each tuple is repeated R_F times to give a *rankings* dataset size of R_FGB per worker. The query scans $R_F \times 72$MB of pageRank data. Figure 15 shows total time and overhead for 10 workers running the scan, as we vary R_F. With $R_F = 200$, we scan 14.4GB of data in 982ms for the scan alone and 1.2secs total including scheduling . As expected, the

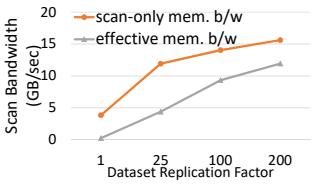

Figure 16: Scan bandwidth, varying dataset size

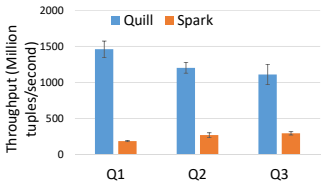

Figure 17: Complex scan queries; varying predicate complexity

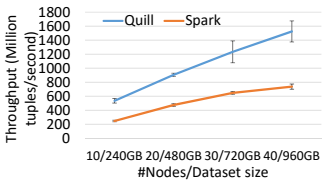

Figure 18: Aggregate query with 2K groups

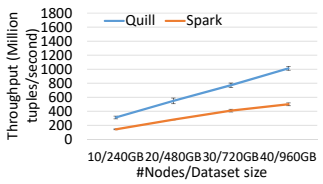

Figure 19: Aggregate query with 67K groups

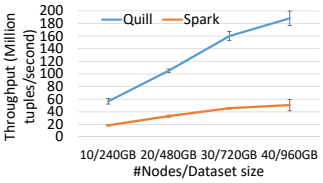

Figure 20: Aggregate query with 40 million groups

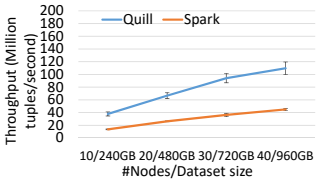

Figure 21: Aggregate query with 140 million groups

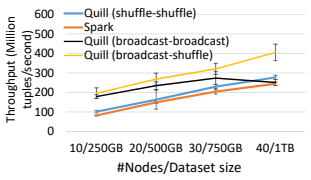

Figure 22: Comparison of different joins; 140M groups

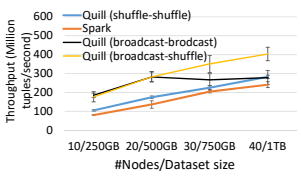

Figure 23: Comparison of different joins; 67K groups

scheduling overhead does not change by much; thus, overhead is a lower fraction of total runtime as we increase R_F. Figure 16 shows scan bandwidth; as the dataset size increases, overhead is lower and the scan reaches 15.6GB/sec per worker (11.9GB/sec if we include scheduling) for $R_F = 200$. This is respectable because the theoretical maximum memory bandwidth of 50GB/sec for these machines is just around 3.2× our measured scan bandwidth.

3) Complex scans. Fig. 17 compares Quill and Spark for three complex scans using the *uservisits* table with 10 workers. *Q1* contains an expensive *LIKE* predicate, where Quill benefits from columnar RegEx matching [15] and achieves throughput 8× higher than Spark. *Q2* and *Q3* contain 2 and 4 predicates (comparisons and a *LIKE* predicate) respectively; Quill is around 4× faster for these queries.

For small data, both systems are dominated by overheads, with Quill scaling better with lower overheads. For larger data, Quill's scan runs at 15.6GB/sec (11.9GB/sec including scheduling). Quill performs better for more complex scans as well.

6.3.2 Grouped Aggregate Queries

1) Aggregate performance. The second query of the benchmark is an *aggregate query* (see below). The query compute the total revenue originated per *sourceIP*. We want to control the complexity of the query by modifying the number of groups. To do so, we modify the length of the *sourceIP* prefix as suggested in the benchmark. We run the query with 2K, 67K, 40M and 140M groups.

```
SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
FROM UserVisits GROUP BY SUBSTR(sourceIP, 1, 7);
```

Executing this query involves a local pre-aggregation, a shuffle across the network to partition by (*sourceIP*), and a final aggregation. Thus, the data needed to be communicated across machines depends on the number of groups. Interestingly, we found that for a small number of groups (2K and 67K), the best strategy involves aggregating per-core before re-keying and re-distributing for the global shuffle. On the other hand, for larger number of groups, the best strategy ended up being first re-keying and re-distributing across cores, aggregating, and then shuffling globally. These strategies correspond to the query plans of Fig. 1(b) and (d), which are easily expressed using the ShardedStreamable API (§3). SparkSQL uses the Catalyst optimizer that chooses plans automatically, and does not provide users the ability to control or fine-tune the plan as needed to optimally execute this experiment.

Fig. 18, 19, 20 and 21 show the results for the query for 2K, 67K, 40M and 140M respectively. The immediate difference with respect to the *scan query* is that these graphs show how both Quill and Spark scale with the cluster size. The reason is that overheads do not dominate the query execution time, and thus more resources lead to a reduced query completion time.

When the number of groups is small, the amount of data that is required to shuffle is small, query time is dominated by the execution of the aggregate function, such as in Fig. 18 and Fig. 19. Quill outperforms Spark by a factor of 2×, due to the optimized physical plan, lower scheduling overheads, and higher operator efficiency. As the number of groups increases, shuffling time increases, which explains the lower performance in both systems. With a higher number of groups, the efficiency gains of Quill over Spark broaden, showing an improvement of around 3.5× (see Fig. 20 and Fig. 21).

2) Aggregate execution time split-up in Quill. With smaller group sizes (2K and 67K), execution is CPU-bound by the per-core pre-aggregation because the dataset is highly reduced after that point. For the larger group size (40M groups), different phases of the query had different bottlenecks; a run in Quill that took 26secs overall spent 38% on per-core aggregation (CPU-bound), 34.6% on intra-machine shuffle and re-aggregation (memory-bound), 22.6% on inter-machine shuffle (network and serialization-bound), and 4.8% on the final aggregation (CPU-bound).

With efficient group-aggregation over columnar batches, fast network communication, low scheduling overheads, the ability to fine-tune the physical plan and avoid extra hash computations, Quill can run aggregate queries at high performance.

6.3.3 Join Queries

The last query of the benchmark is the *join query* shown below. The purpose of the query is to calculate the *sourceIP* and associated *pageRank* that gave rise to the highest revenue for a given period of time, e.g. 1 week. We vary the complexity of this query by changing the total number of groups according to the *sourceIP* attribute, as in the case of the *aggregate query*.

```
SELECT INTO Temp sourceIP, AVG(pageRank) as avgPageRank,
                  SUM(adRevenue) as totalRevenue
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL
AND UV.visitDate BETWEEN Date('2000-01-15')
                     AND Date('2000-01-22')
GROUP BY UV.sourceIP;
SELECT sourceIP, totalRevenue, avgPageRank FROM Temp
ORDER BY totalRevenue DESC LIMIT 1;
```

We implement the query using SparkSQL's physical plan. This plan first applies the time-range filter to reduce the *UserVisits* dataset, after which both datasets are shuffled on the join key, which is in this case *pageURL* and *destURL*. After the shuffle and join, the data

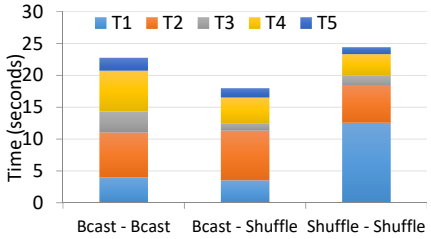

Figure 24: Effect of modifying physical plan for different join strategies

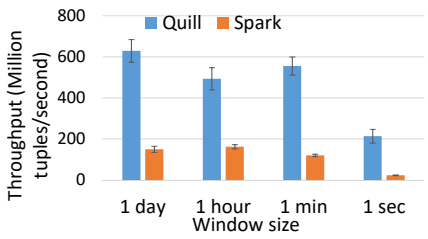

Figure 25: Performance of tumbling window query for different window sizes

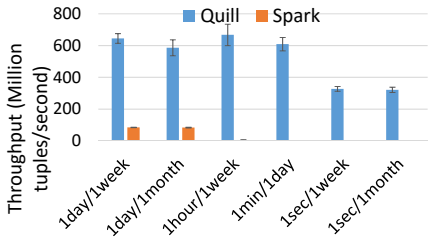

Figure 26: Performance of hopping window query for different window slide/size values

is shuffled again, this time on *sourceIP*, before running the final aggregate and top-K. We call this plan *shuffle-shuffle* to denote that a shuffle occurs across machines and across cores in each machine before the join (Fig. 4(a) shows this join strategy).

Fig. 22 and Fig. 23 show the results for the *join query* when run with both 140M and 67K groups. For shuffle-shuffle, the performance of both Quill and Spark is mostly influenced by the time it takes to transmit the datasets through the network. The slightly better throughput of Quill in this case is likely due to more efficient query processing and data movement.

We then implemented two additional plans for JOIN, that are more efficient than the one chosen by the Spark query optimizer. We run a strategy called *Broadcast-Broadcast*, depicted in Fig. 4(b), that does not shuffle the *Rankings* dataset. Instead, it broadcasts the filtered *UserVisits* dataset to all machines in the cluster, and then to each core on each machine. Fig. 22 and Fig. 23 show how this strategy is faster than naively shuffling both datasets. However, the performance of broadcast-broadcast suffers when the dataset size grows beyond 750 GB. The reason is that in this strategy, each core needs to build a hash table over the entire dataset, which grows linearly with the number of nodes. A better strategy is therefore *broadcast-shuffle*, whose plan is depicted in Fig. 4(c). In this case, we perform a broadcast across machines to avoid the expensive shuffle of *Rankings*, but then we perform shuffle both datasets locally, which partitions the hash table across cores, bringing additional performance benefits, as shown in Fig. 22 and Fig. 23.

Fig. 24 shows how time is spent in the case of the three join strategies. *T1* and *T2* are the time spent in applying the filter, rekeying and redistributing the *Rankings* and *UserVisits* tables respectively. This clearly shows how shuffling *Rankings* table is more expensive than broadcasting *UserVisits*. *T3* is the time spent performing the join operation, while *T4* is the time spent aggregating page rank and ad revenue per source IP. We see that the broadcast-broadcast strategy is more expensive due to the higher cost of maintaining a hash table per core. Finally *T5* is the time spent on Top-K.

By expressing a large physical plan space, Quill can run join queries on large datasets with high performance.

6.4 Temporal Analytical Queries

We next evaluate temporal queries that compute various grouped aggregates over *tumbling* and *hopping (sliding)* windows, on a cluster with 10 machines. We vary the window size, and the slide in case of hopping windows. We also run the queries on SparkSQL, since time is a column in the GitHub schema, to compare performance.

1) Tumbling window query. Tumbling windows are easy to express in SparkSQL, by grouping by the (coarsened) time field. The results are shown in Fig. 25. Quill has a performance 3×-4× higher than Spark. Both systems show a similar behavior for 1 day, 1 hour and 1 min windows. Performance reduces severely in the case of 1 second windows for both systems. Quill becomes 2× slower, while Spark performance drops by a factor of 8. The reasons for this decrease in performance, however, differ between systems. Quill's performance reduces due to the higher amount of data generated, i.e. the bottleneck is not query processing, but data delivery. Since Spark groups by time, query processing suffers from a higher number of groups created due to the smaller window size.

2) Hopping window query. The *hopping* window query is more demanding for SparkSQL; while it can leverage the panes trick [24] to implement hopping windows, the number of groups necessary to maintain grows. We could only run the first three configurations of the query, window slide/size of 1 day/1 week, 1 day/1 month, and 1 hour/1 week. Other configurations took too long or did not complete, and we do not include them. The results of Fig. 26 show that the cost for Quill does not change—computing a hopping window is similar to a tumbling window, due to native temporal support. Quill's throughput is up to 120× higher than Spark (for queries that we were able to complete on Spark). As before, throughput reduces for a window slide of 1 second due to the cost of delivering results.

By carefully layering a temporal library and adding efficient ReKey, ReDistribute, and time-ordered merge operations, Quill can process temporal queries very efficiently.

6.5 Shuffle and Data Loading Performance

We often need to process raw data in timestamp order by key, for example, to detect sequential patterns in the data. Such queries may not benefit from pre-aggregation, resulting in the need to shuffle the raw data. We generate a synthetic dataset with two random long fields, and execute a `Broadcast` and a shuffle (`ReKey+ReDistribute`) query (across all machines). For broadcast, we found the actual shuffle throughput (incoming plus outgoing) to reach 9.79 Gbps per node on a 40-machine cluster, which is close to the 10 Gbps NIC bandwidth limit. Shuffle additionally involves computing hash values and moving data by hash bucket before sending it on the wire; we achieve an effective throughput of 8.35 Gbps per node. Note that users could also materialize datasets after a `ReKey` to leverage pre-computed keys and hash values.

Next, we measure the performance of loading a 762GB comma-separated text dataset with 16 fields, stored across 8 Azure storage accounts. With 40 cloud workers, we load the data is 85secs, with an average read throughput of around 9 Gbps per storage account, close to the enforced limit of 10 Gbps per storage account.

7. RELATED WORK

In Section 1.1 and Table 1, we covered today's analytics platforms in terms of satisfying our target requirements; more details follow.

Databases & big data systems. DBMSs compile SQL queries into plans with high performance, but the plans cannot be modified, programmed, or transferred to real-time. They lack temporal operator support or deep integration into the HLL type-system. Beyond broadcast and exchange operators, commercial DBMSs do not offer more complex data-dependent duplication strategies for data movement. As Fig. 2 shows, Quill offers an alternative to query

hinting that allows users to control and validate their plans, or program them directly in a HLL. By making it easy for optimizers to express and execute plans, we expect to see the growth of *HLL optimizer libraries* that target specific verticals or scenarios as well. Systems such as map-reduce [19] and Spark RDDs [33] expose a keyed computation model to exploit data-parallelism, but are unable to express and optimize for the rich space of distributed execution strategies we target. These systems also expose SQL front-ends such as Hive [31] that have limitations as discussed in Section 1. In contrast, Quill separates query logic, data movement, and logical keying to allow the expression of a rich space of physical plans that implement complex temporal logic and are transferable to real-time as well. Moreover, existing systems implement a master-worker architecture unlike Quill, which uses decentralized cloud structures for client-driven coordination.

Streams & Physical Algebras. Single-node streaming systems [9, 15] have rich query models but are inadequate to meet high data volumes and scale. Streaming systems such as Storm [7], Flink [4], MillWheel [2], and Google Cloud Dataflow [3] expose a fine-grained API using which users can process and produce ⟨key, value⟩ pairs, with richer temporal APIs on top. As discussed in §3.3.2, the fine-grained API results in several inefficiencies at the data plane. In contrast, Quill provides functional transformations such as ReKey, ReDistribute, and Multicast that are executed at the data plane in batched columnar form by exploiting white-box lambda expression parameters and code generation. Dryad's physical algebra is an arbitrary graph which cannot be programmed, unlike Quill which exposes a small, readable, and powerful set of functional transformations to specify physical plans. DryadLINQ and SparkSQL operate at the logical specification level like databases, and users cannot tweak physical plans. Quill's physical algebra explicitly separates data movement from keying, and adds new data movement functions beyond ReDistribute that expand query expressiveness without giving up on performance. Further, existing systems do not support primitives to optimize inter-node vs. intra-node plans. Finally, the separation of `Query` as a layer allows temporal logic specification (leveraging Trill) independent from dataflow specification, guaranteeing transferability without loss of performance, by exposing batching and grouping to the innermost loop.

8. CONCLUSIONS

Quill is a library and distributed platform for analytics over large datasets. We propose a new abstraction called *ShardedStreamable*, that can express efficient distributed physical query plans that are transferable between real-time and offline deployments ShardedStreamable decouples incremental query logic specification, data movement, and keying; this allows Quill to express a broad space of plans with complex querying functionality, while leveraging temporal libraries such as Trill. Quill's layered architecture provides a separation of responsibilities with independently useful components, while retaining high performance. We built Quill for the cloud, with a master-less design that leverages off-the-shelf distributed components found in cloud providers. Experiments on up to 400 cloud machines, on benchmark and real datasets up to 1TB, find Quill to incur low overhead and outperform SparkSQL by up to orders-of-magnitude for temporal and 6× for relational queries, while supporting significantly expanded temporal querying functionality.

Acknowledgments. We would like to thank Mike Barnett, Tyson Condie, Ravi Ramamurthy, James F. Terwilliger, Markus Weimer, and the anonymous reviewers for their comments, help, and support.

9. REFERENCES

[1] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.
[2] T. Akidau et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *VLDB*, 2013.
[3] T. Akidau et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. In *VLDB*, 2015.
[4] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.
[5] Apache. Calcite. `http://calcite.apache.org`.
[6] Apache Kafka. `http://kafka.apache.org/`.
[7] Apache Storm. `http://storm.apache.org/`.
[8] M. Armbrust et al. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.
[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. *PODS*, 2002.
[10] S. Blanas et al. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, 2010.
[11] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power hints for query optimization. In *ICDE*, 2009.
[12] B. Chandramouli et al. The Quill Distributed Analytics Library and Platform. Technical report, Microsoft Research (MSR-TR-2016-25). `http://aka.ms/quill-tr`.
[13] B. Chandramouli et al. StreamRec: A Real-Time Recommender System. 2011.
[14] B. Chandramouli et al. Scalable Progressive Analytics on Big Data in the Cloud. In *VLDB*, 2014.
[15] B. Chandramouli et al. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.*, 8(4):401–412, Dec. 2014.
[16] B. Chandramouli, J. Goldstein, and S. Duan. Temporal Analytics on Big Data for Web Advertising. In *ICDE*, 2012.
[17] S. Chu et al. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *SIGMOD*, 2015.
[18] Cloudera Engineering Blog. Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real. `http://tinyurl.com/bsrhcf7`.
[19] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *CACM*, 2008.
[20] Forcing Query Plans in SQL Server. `http://aka.ms/etgyp4`.
[21] GitHub Archive. `https://www.githubarchive.org/`.
[22] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, 18, 1995.
[23] M. Kornacker et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.
[24] J. Li et al. No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams. *SIGMOD Rec.*, 2005.
[25] N. Marz. How to beat the CAP theorem. `http://tinyurl.com/3lqwebc`.
[26] Expression Trees in .NET. `https://msdn.microsoft.com/en-us/library/bb397951.aspx`.
[27] Microsoft Azure. `https://azure.microsoft.com/en-us/`.
[28] A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In *SIGMOD*, 2011.
[29] Oracle9i Database Performance Tuning Guide and Reference. Using Explain Plan. `http://tinyurl.com/hkh7zx6`.
[30] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.
[31] A. Thusoo et al. Hive - A Warehousing Solution Over a Map-Reduce Framework. *VLDB*, 2009.
[32] R. S. Xin, J. Rosen, et al. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.
[33] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.