Cheetah: Accelerating Database Queries with Switch Pruning

Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, Minlan Yu
Harvard University
{tirmazi,ran,jiaqigao,minlanyu}@g.harvard.edu

Abstract
Modern database systems are growing increasingly distributed and struggle to reduce query completion time with a large volume of data. In this paper, we leverage programmable switches in the network to partially offload query computation to the switch. While switches provide high performance, they have resource and programming constraints that make implementing diverse queries difficult. To fit in these constraints, we introduce the concept of data pruning – filtering out entries that are guaranteed not to affect output. The database system then runs the same query but on the pruned data, which significantly reduces processing time. We propose pruning algorithms for a variety of queries. We implement our system, Cheetah, on a Barefoot Tofino switch and Spark. Our evaluation on multiple workloads shows 40 – 200% improvement in the query completion time compared to Spark.

1 Introduction
Database systems serve as the foundation for many applications such as data warehousing, data analytics, and business intelligence [49]. Facebook is reported to run more than 30,000 database queries that scan over a petabyte per day [24]. With the increase of workloads, the challenge for database systems today is providing high performance for queries on a large distributed set of data.

A popular database query processing system is Spark SQL [36]. Spark SQL optimizes query completion time by assigning tasks to workers (each working on one data partition) and aggregating the query result at a master worker. Spark maximizes the parallelism and minimizes the data movement for query processing. For example, each worker sends a stream of the resulting metadata (e.g., just the columns relevant to the query) before sending the entire rows that are requested by the master. Despite the optimizations, the query performance is still limited by software speed.

We propose Cheetah, a query processing system that partially offloads queries to programmable switches. Programmable switches are supported by major switch vendors [10, 11, 39, 40]. They allow programmatic processing of multiple Tbps of traffic [39], which is orders of magnitude higher throughput than software servers and alternative hardware such as FPGAs and GPUs. Moreover, switches already sit between the workers and thus can process aggregated data across partitions.

However, we cannot simply offload all database operations to switches as they have a constrained programming model [9]: Switches process incoming packets in a pipeline of stages. At each stage, there is a limited amount of memory and computation. Further, there is a limited number of bits we can transfer across stages. These constraints are at odds with the large amount of data, diverse query functions, and many intermediate states in database systems.

To meet these constraints, we propose a new abstraction called pruning. Instead of offloading full functionality to programmable switches, we use the switch to prune a large portion of data based on the query, and the master only needs to process the remaining data in the same way as it does without the switch. For example, the switch may remove some duplicates in a DISTINCT query, and let the master remove the rest, thus accelerating the query.

The pruning abstraction allows us to design algorithms that can fit with the constrained programming model at switches: First, we do not need to implement all database functions on the switches but only offload those that fit in the switch’s programming model. Second, to fit in the limited memory of switches, we either store a cached set of results or summarized intermediate results while ensuring a high pruning rate. Third, to reduce the number of comparisons, we use in-switch partitioning of the data such that each entry is only compared with a small number of entries in its partition. We also use projection techniques that map high-dimensional data points into scalars which allows an efficient comparison.

Based on the pruning abstraction, we design and develop multiple query algorithms ranging from filtering and DISTINCT to more complex operations such as JOIN or GROUP BY. Our solutions are rigorously analyzed and we prove bounds on the resulting pruning rates. We build a prototype on the Barefoot Tofino programmable switch [39] and demonstrate 40 – 200% reduction of query completion times compared with Spark SQL.

2 Using programmable switches
Cheetah leverages programmable switches to reduce the amount of data transferred to the query server and thus
improves query performance. Programmable switches that follow the PISA model consist of multiple pipelines through which a network packet passes sequentially. These pipelines contain stages with disjoint memory which can do a limited set of operations as the packet passes through them. See [46] for more information about the limitations. In this section, we discuss the benefits and constraints of programmable switches to motivate our pruning abstraction.

2.1 Benefits of programmable switches

We use Spark SQL as an example of a database query execution engine. Spark SQL is widely used in industry [36], adopts common optimizations such as columnar memory-optimized storage and vectorized processing, and has comparable performance to Amazon’s RedShift [18] Google’s BigQuery [23].

When a user submits a query, Spark SQL uses a Catalyst optimizer that generates the query plan and operation code (in the form of tasks) to run on a cluster of workers. The worker fetches a partition from data sources, runs its assigned task, and passes the result on to the next batch of workers. At the final stage, the master worker aggregates the results and returns them to the user. Spark SQL optimizes query completion time by having workers process data in their partition as much as possible and thus minimizes the data transferred to the next set of workers. As a result, the major portion of query completion time is spent at the tasks the workers run. Thus, Spark SQL query performance is often bottlenecked by the server processing speed and not the network [36].

Cheetah rides on the trend of significant growth of the network capacity (e.g., up to 100Gbps or 400Gbps per port) and the advent of programmable switches, which are now provided by major switch vendors (e.g., Barefoot [39, 40], Broadcom [10], and Cavium Xpliant [11]). These switches can process billions of packets per second, already exist in the data path, and thus introduce no latency or additional cost. Table 3 compares the throughput and delay of Spark SQL on commodity servers with those of programmable Tofino switches. The best throughput of servers is 10-100Gbps, but switches can reach 6.5-12.8 Tbps. The switch throughput is also orders of magnitudes better than alternative hardware choices such as FPGAs, GPUs, and smart NICs. Switches also have less than 1 μs delay per packet.

These switches already exist in the cluster and already see the data transferred between the workers. They are at a natural place to help process queries. By offloading part of the processing to switches, we can reduce the workload at workers and thus significantly reduce the query completion time despite more data transferred in the network. Compared with server-side or storage-side acceleration, switches have the extra benefit in that it can process the aggregated data across workers. We defer the detailed comparison of Cheetah with alternative hardware solutions to §10.

2.2 Constraints of programmable switches

Programmable switches make it possible to offload part of queries because they parse custom packet formats and thus can understand data block formats. Switches can also store a limited amount of state in their match-action tables and make comparisons across data blocks to see if they match a given query. However, there are several challenges in implementing queries on switches:

**Function constraints:** There are limited operations we can run on switches (e.g. hashing, bit shifting, bit matching, etc). These are insufficient for queries which sometimes require string operations, and other arithmetic operations (e.g., multiplication, division, log) on numbers that are not power-of-twos.

**Limited pipeline stages and ALUs:** Programmable switches use a pipeline of match action tables. The pipeline has a limited number of stages (e.g. 12-60) and a limited number of ALUs per stage. This means we can only do a limited number of computations at each stage (e.g. no more than ten comparisons in one stage for some switches). This is not enough for some queries which require many comparisons across entries (e.g. DISTINCT) or across many dimensions (e.g. SKYLINE).

**Memory and bit constraints:** To reach high throughput and low latency, switches have a limited amount of on-chip memory (e.g. under 100MB of SRAM and up to 100K-300K TCAM entries) that is partitioned between stages. However, if we use switches to run queries, we have to store, compare, and group a large number of past entries that can easily go beyond the memory limit. Moreover, switches only parse a limited number of bits and transfer these bits across stages (e.g., 10-20 Bytes. Some queries may have more bits for the keys especially when queries are on multiple dimensions or long strings.

3 Cheetah design

**The pruning abstraction:** We introduce the pruning abstraction for partially offloading queries onto switches. Instead of offloading the complete query, we simply offload a critical part of a query. In this way, we best leverage the high throughput and low latency performance of switches while staying within their function and resource constraints. With pruning, the switch simply filters the data sent from the workers, but does not guarantee query completion. The master runs queries on the pruned dataset and generates the same query result as if it had run the query with the original dataset. Formally, we define pruning as follows; Let $Q(D)$ denote the result (output) of query $Q$ when applied to an input data $D$. A pruning algorithm for $Q$, $A_Q$, gets $D$ produces $A_Q(D) \subseteq D$ such that $Q(A_Q(D)) = Q(D)$. That is,
the algorithm computes a subset of the data such that the output of running $Q$ on the subset is equivalent to that of applying the query to the whole of $D$.

To make our description easier, for the rest of the paper, we focus on queries with one stage of workers and one master worker. We also assume a single switch between them. An example is a rack-scale query framework where all the workers are located in one rack and the top-of-rack switch runs Cheetah pruning solutions. Our solutions can work with multiple stages of workers by having the switch prune data for each stage. We discuss how to handle multiple switches in §9.

**Cheetah architecture:** Cheetah can be easily integrated within Spark without affecting its normal workflow. Figure 1 shows the Cheetah design:

**Query planner:** The way users specify a query in Spark remains unchanged. For example, the query (e.g., `SELECT * WHERE c > θ`) has three parameters: (1) the query type (filtering in this example), (2) the query parameters $θ$, and (3) the relevant columns ($c$ in this example). In addition to distributing tasks to workers, the query planner sends (1) and (2) to the switch control plane which updates the switch memory accordingly. Once the master receives an ACK from the switch, which acknowledges that it is ready, it starts the tasks at workers.

**CWorkers:** With Cheetah, the workers do not need to run computationally intensive tasks on the data. Instead, we implement a Cheetah module (CWorker), which intercepts the data flow at workers and sends the data directly through the switch. Therefore, Cheetah reduces the processing time at the workers and partially offloads their tasks to the switch. The workers and the master only need to process the remaining part of the query. CWorkers also convert the related columns into packets that are readable by the switch. For example, if some entries have variable width or are excessively wide (e.g., a DISTINCT query on multiple columns), CWorkers may compute fingerprints before sending the data out.

**Cheetah Switch:** Cheetah switch is the core component of our system. We pre-compile all the common query algorithms at the switch. At runtime, when the switch receives a query and its parameters, it simply installs match-action rules based on the query specifications. Since most queries just need tens of rules, the rule installation takes less than 1 ms in our evaluation. According to these rules, the switch prunes incoming packets by leveraging ALUs, match-action tables, registers, and TCAM as explained in §4. The switch only forwards the remaining packets to the CMaster. The switch identifies incoming packets from workers based on pre-specified port numbers. This allows the switch to be fully compatible with other network functions and applications sharing the same network. Since the switch only adds acceleration functions by pruning the packets, the original query pipeline can work without the switch. If the switch fails, operators can simply reboot the switch with empty states or use a backup ToR switch. We also introduce a new communication protocol, as explained in detail in §7.2, that allows the workers to distinguish between pruned packets that are legitimately dropped and lost packets that should be retransmitted.

**CMaster:** At the master, we implement a Cheetah module (CMaster) that converts the packets back to original Spark data format. The Spark master works in the same way with and without Cheetah. It “thinks” that we are running the query on the pruned dataset rather than the original one and completes the operation. As the pruned dataset is much smaller, the Spark master takes less time to complete on Cheetah. Many Spark queries adopt late materialization: Spark first runs queries on the metadata fields (i.e., those columns of an entry that the query conditions on) and then fetches all the requested columns for those entries that match the criteria. In this case, Cheetah prunes data for the metadata query and does not modify the final fetches.

### 4 Query Pruning Algorithms

In this section, we explore the high-level primitives used for our query pruning algorithms. An example of input tables which we will use to illustrate them is given in Table 1. We also provide a table summarizing our algorithms, their parameters and pruning guarantee type in the full version [50].

<table>
<thead>
<tr>
<th>(a) Products</th>
<th>(b) Ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td><strong>seller</strong></td>
</tr>
<tr>
<td>Burger</td>
<td>McCheetah</td>
</tr>
<tr>
<td>Pizza</td>
<td>Papizza</td>
</tr>
<tr>
<td>Fries</td>
<td>McCheetah</td>
</tr>
<tr>
<td>Jello</td>
<td>JellyFish</td>
</tr>
<tr>
<td>Fries</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: Running Database Example
4.1 Handling Function Constraints

Due to the switch’s function and resource limitations, we cannot always prune a complete query. In such cases, Cheetah automatically decomposes the query into two parts and prunes parts of the query which are supported.

**Example #1: Filtering Query:**

Consider the common query in database of selecting entries matching a WHERE expression, for example:

```
SELECT * FROM Ratings WHERE (taste > 5)
OR (texture > 4 AND name LIKE e%)
```

The switch may not be able to compute some expressions due to lack of function support (e.g., if it cannot evaluate `name LIKE e%`) or may lack ALUs or memory to compute some functions. Cheetah runs a subset of the predicates at the switch to prune the data and runs the remaining either at the workers or the master.

The Cheetah query compiler decomposes the predicates into two parts: Consider a monotone Boolean formula \( \phi = f(x_1, \ldots, x_n, y_1, \ldots, y_m) \) over binary variables \( \{x_i\}, \{y_i\} \) and assume that the predicates \( \{x_i\} \) can be evaluated at the switch while \( \{y_i\} \) cannot. The Cheetah query compiler replaces each variable \( y_i \) with a tautology (e.g., \((T \lor F)\)) and applies standard reductions (e.g., modus ponens) to reduce the resulting expression. The resulting formula is computable at the switch.

In our example, we transform the query into:

\[
(taste > 5) \lor (texture > 4 \land (T \lor F)) \equiv \neg(taste > 5) \lor (texture > 4) 
\]

Therefore, Cheetah prunes entries that do not satisfy \((taste > 5) \lor (texture > 4)\) and let the master node complete the operation by removing entries for which \(\neg(taste > 5) \land (texture > 4)\). In other cases, Cheetah uses workers to compute the predicates that cannot be evaluated on the switch. For instance, the CWorker can compute \((name \text{ LIKE e%})\) and add the result as one of the values in the sent packet. This way the switch can complete the filtering execution as all predicate values are now known.

Cheetah supports combined predicates by computing the basic predicates that they contain \((taste > 5 \land texture > 4)\) and then checking the condition based on the true/false result we obtain for each basic predicate. Cheetah writes the values of the predicates as a bit vector and looks up the value in a truth table to decide whether to drop or forward the packets.

4.2 Handling Stage/ALU Constraints

Switches have limited stages and limited ALUs per stage. Thus, we cannot compare the current entry with a sufficiently large set of points. Fortunately, for many queries, we can partition the data into multiple rows such that each entry is only compared with those in its row. Depending on the query, the partitioning can be either randomized or hash-based as in the following.

**Example #2: DISTINCT Query:**

The DISTINCT query selects all the distinct values in an input columns subset, e.g.,

```
SELECT DISTINCT seller FROM Products
```

returns (Papizza, McCheetah, JellyFish). To prune DISTINCT queries, the idea is to store all past values in the switch. When the switch sees a new entry, it checks if the new value matches any past values. If so, the switch prunes this entry; if not, the switch forwards it to the master node. However, storing all entries on the switch may take too much memory.

To reduce memory usage, an intuitive idea is to use Bloom filters (BFs) [7]. However, BFs have false positives. For DISTINCT, this means that the switch may drop entries even if their values have not appeared. Therefore, we need a data structure that ensures no false positives but can have false negatives. Caches match this goal. The master can then remove the false negatives to complete the execution.

We propose to use a \(d \times w \) matrix in which we cache entries. Every row serves as a Least Recently Used (LRU) cache that stores the last \(w\) entries mapped to it. When an entry arrives, we first hash it to \(\{1, \ldots, d\}\), so that the same entry always maps to the same row. Cheetah then checks if the current entry’s value appears in the row and if so prunes the packet. To implement LRU, we also do a rolling replacement of the \(w\) entries by replacing the first one with the new entry, the second with the first, etc. By using multiple rows we reduce the number of per-packet comparisons to allow implementation on switches.

In the full version [50], we analyze the pruning ratio on random order streams. Intuitively, if row \(i\) sees \(D_i\) distinct values and each is compared with \(w\) that are stored in the switch memory, then with probability at least \(w/D_i\), we will prune every duplicate entry. For example, consider a stream that contains \(D = 15000\) distinct entries and we have \(d = 1000\) rows and \(w = 24\) columns. Then we are expected to prune \(58\%\) of the duplicate entries (i.e., entry values that have appeared previously).

**Theorem 1.** Consider a random order stream with \(D > d \ln(200d)\) distinct entries\(^1\). Our algorithm, configured with \(d\) rows and \(w\) columns is expected to prune at least \(0.99 \cdot \min\left\{ \frac{D}{d}, 1 \right\}\) fraction of the duplicate entries.

4.3 Handling Memory Constraints

Due to switch memory constraints, we can only store a few past entries at switches. The key question is: how do we

\(^1\)It’s possible to optimize other cases, but this seems to be the common case.
may return (Jello 4, Cheetos 6, Pizza 5). Pruning in a TOP N query means that we may return a superset of the largest N values. The intuitive solution is to store the N largest values, one at each stage. We can then compare them and maintain a rolling minimum of the stages. However, when N is much larger than the number of stages (say N is 100 or 1000 compared to 10-20 stages), this approach does not work.

Instead, we use a small number of threshold-based counters to enable pruning TOP N query. The switch first computes the minimal value $t_0$ for the first N entries. Afterward, the switch can safely filter out everything smaller than $t_0$. It then tries to increase the threshold by checking how many values larger than $t_0$ we observe, for some $t_1 > t_0$. Once N such entries were processed, we can start pruning entries smaller than $t_1$. We can then continue with larger thresholds $t_2, t_3, \ldots, t_w$. We set the thresholds exponentially ($t_i = 2^i \cdot t_0$) in case the first N is much smaller than most in the data. This power-of-two choice also makes it easy to implement in switch hardware. When using w thresholds, our algorithm can get pruning points smaller than $t_0 \cdot 2^{w-1}$, if enough larger ones exist.

Example #4: JOIN Query:
In a JOIN operation we combine two tables based on the target input columns.\(^3\) For example, the query

```
SELECT Products.name FROM Products
JOIN Ratings
ON Products.name = Ratings.name
```

gives

<table>
<thead>
<tr>
<th>name</th>
<th>seller</th>
<th>price</th>
<th>taste</th>
<th>texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burger</td>
<td>McCheeta</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Pizza</td>
<td>Papizza</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Fries</td>
<td>McCheeta</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Jello</td>
<td>JellyFish</td>
<td>5</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

In the example, we can save computation if the switch identifies that the key "Cheetos" did not appear in the `Products` table and prune it. To support JOIN, we propose to send the data through the switch with two passes. In the first pass, we use Bloom Filters [7] to track observed keys. Specifically, consider joining tables A and B on input column (or input columns) C. Initially, we allocate two empty Bloom filters $F_A, F_B$ to approximately record the observed values (e.g., `name`) by using an input column optimization to stream the values of C from both tables. Whenever a key $x$ from table A (or B) is processed on the switch, Cheetah adds $x$ to $F_A$ (or $F_B$). Then, we start a second pass in which the switch prunes each packet $x$ from A (respectively, B) if $F_B$ ($F_A$) did not report a match. As Bloom Filters have no false negatives, we are guaranteed that Cheetah does not prune any matched entry. In the case of JOIN, the false positives in Bloom Filters only affect the pruning ratio while the correctness is guaranteed. Such a two pass strategy causes more network traffic but it significantly reduces the processing time at workers.

If the joined tables are of significantly different size, we can optimize the processing further. We first stream the small table without pruning while creating a Bloom filter for it. Since it is smaller, we do not lose much by not pruning and we can create a filter with significantly lower false positive rate. Then, we stream the large table while pruning it with the filter.

Example #5: HAVING Query:
HAVING runs a filtering operation on top of an aggregate function (e.g., MIN/MAX/SUM/COUNT). For example,

```
SELECT seller FROM Products GROUP BY seller
HAVING SUM(price) > 5
```

should return (McCheeta, Papiza). We first check the aggregate function on each incoming entry. For MAX and MIN, we simply maintain a counter with the current max and min value. If it is satisfied, we proceed to our Distinct solution (see §4.2) – if it reports that the current key has not appeared before, we add it to the data structure and forward the entry; otherwise we prune it.

SUM and COUNT are more challenging because a single entry is not enough for concluding whether we should pass the entry. We leverage sketches to store the function values for different entries in a compact fashion. We choose Count-Min sketch instead of Count sketch or other algorithms because Count-Min is easy to implement at switches and it has one-sided error. That is, for HAVING $f(x) > c$, where $f$ is SUM or COUNT, Count-Min gives an estimator $f(z)$ that satisfies $f(z) \geq f(x)$. Therefore, by pruning only if $f(z) \leq c$ we guarantee that every key $x$ for which $f(x) > c$ makes it to the master. Thus, the sketch estimation error only affects the pruning rate. After the sketch, the switches blocks all the traffic to the master. Then we make a partial second pass (i.e., stream the data again), only for the keys requested by the master. That is, the master gets a superset of the keys

\(^2\) In different systems this operation have different names; e.g., MySQL supports LIMIT while Oracle has ROWNUM.

\(^3\) We refer here to INNER JOIN, which is SQL’s default. With slight modifications, Cheetah can also prune LEFT/RIGHT OUTER joins.
that it should output and requests all entries that belong to
them. It can then compute the correct COUNT/MAX and
remove the incorrect keys (whose true value is at most c).
We defer the support for \(\text{SUM/COUNT} < c\) operations to
future work.

4.4 Projection for High-dimensional Data

So far we mainly focus on database operations on one dimen-
sion. However, some queries depend on values of multiple
dimensions (e.g., in SKYLINE). Due to the limited number
of stages and memory at switches, it is not possible to store
and compare each dimension. Therefore, we need to project
the multiple dimensions to one value (i.e., a fingerprint). The
normal way of projection is to use hashing, which is useful
for comparing if an entry is equal to another (e.g., DISTINCT
and JOIN). However, for other queries (e.g., SKYLINE), we
may need to order multiple dimensions so we need a different
projection strategy to preserve the ordering.

Example #6: SKYLINE Query:
The Skyline query [8] returns all points on the Pareto-curve
of the \(D\)-dimensional input. Formally, a point \(x\) is dominated
by \(y\) only if it is dominated on all dimensions, i.e., \(\forall i \in \{1, \ldots, D\} : x_i \leq y_i\). The goal of a SKYLINE query is to
find all the points that are not dominated in a dataset.\(^4\) For
example, the query

```sql
SELECT name FROM Ratings
WHERE taste = Cheetos, texture = Jello, Burger.
```

Because skyline relates to multiple dimensions, when we
decide whether to store an incoming entry at the switch, we
have to compare with all the stored entries because there
is no strict ordering among them. For each entry, we have to
compare all the dimensions to decide whether to replace
it. But the switch does not support conditional write under
multiple conditions in one stage. These constraints make it
challenging to fit SKYLINE queries on the switch with a
limited number of stages.

To address this challenge, we propose to project each
high-dimensional entry into a single numeric value. We de-
fine a function \(h : \mathbb{R}^D \rightarrow \mathbb{R}\) that gives a single score to
\(D\)-dimensional points. We require that \(h\) is monotonically
increasing in all dimensions to ensure that if \(x\) is dominated
by \(y\) then \(h(x) \leq h(y)\). In contrast, \(h(x) \leq h(y)\) does not
imply that \(x\) is dominated by \(y\). For example, we can define
\(h\) to be the sum or product of coordinates.

The switch stores a total of \(w\) points in the switch. Each
point \(y_{ij}, i = 1 \ldots w\) takes two stages: one for \(h(y_{ij})\) and
another for all the dimensions in \(y_{ij}\). When a new point \(x\)
arrives, for each \(y_{ij}\), the switch first checks if \(h(x) > h(y_{ij})\).

If so, we replace \(h(y_{ij})\) and \(y_{ij}\) by \(h(x)\) and \(x\). Otherwise,
we check whether \(x\) is dominated by \(y_{ij}\) and if so mark \(x\)
for pruning without changing the state. Note that here our
replace decision is only based on a single comparison (and
thus implementable at switches); our pruning decision is
based on comparing all the dimensions but the switch only
drops the packet at the end of the pipeline (not same stage
action).

If \(x\) replaced some \(y_{ij}\), we put \(y_{ij}\) in the packet and con-
tinue the pipeline with the new values. We use a rolling
minimum (according to the \(h\) values) and get that the \(w\)
points stored in the switch are those with the highest \(h\) value
so far are among the true skyline.

The remaining question is which function \(h\) should be.
Product (i.e., \(h_P(x) = \prod_{i=1}^{D} x_i\)) is better than sum (i.e., \(h_S(x) = \sum_{i=1}^{D} x_i\)) because sum is biased towards the \(i\)th
dimension with a larger range (Consider one dimension ranges between
0 and 256 and another between 0 and 65535). However, pro-
duction is hard to implement on switches because it requires
large values and multiplication. Instead, we use Approx-
imate Product Heuristic (APH) which uses the monotonicity
of the logarithm function to represent products as sum-of-
logarithms and uses the switch TCAM and lookup tables to
approximate the logarithm values (see more details in the
full version [50]).

5 Pruning w/ Probabilistic Guarantees

The previous section focuses on providing deterministic
guarantees of pruning, which always ensures the correct-
ness of the query results. Today, to improve query execu-
tion time, database systems sometimes adopt probabilistic
guarantees (e.g., [22]). This means that with a high proba-
bility (e.g., 99.99%), we ensure that the output is exactly as
expected (i.e., no missing entries or extra entries). That is,
\(\Pr[\text{Q(A}_{\text{Q}(D)}) \neq Q(D)] \leq \delta\), where \(Q\) is the query, \(A_Q\)
is the algorithm, and \(D\) is the data, as in §3. Such probabilistic
guarantees allow users to get query results much faster.\(^5\)

By relaxing to probabilistic guarantees, we can improve
the pruning rate by leveraging randomized algorithms to
select the entries to store at switches and adopting hashing
to reduce the memory usage.

Example #7: (Probabilistic) TOP N Query:

Our randomized TOP N algorithm aims at, with a high proba-
bility, returning a superset of the expected output (i.e., none
of the \(N\) output entries is pruned). Cheetah randomly parti-
tions the entries into rows as explained in §4.2. Specifically,
when an entry arrives, we choose a random row for it in
\(\{1, \ldots, d\}\). In each row, we track the largest \(w\) entries mapped
to it using a rolling minimum. That is, the largest entry in
the row is first, then the second, etc. We choose to prune any

\(^4\)For simplicity, we consider maximizing all dimensions. We can extend the
solution to support minimizing all dimensions with small modifications.

\(^5\)The master can check the extra entries before sending the results to users.
Spark can also return the few missing entries at a later time.
entry that was smaller than all \( w \) entries that were cached in its row. Cheetah leverages the balls and bins framework to determine how to set the dimensions of the matrix \((d, w)\) given \( N \), the goal probability \( \delta \), and the resource constraints (the number of stages limits \( w \) while the per-stage memory restricts \( d \)).

The algorithm is illustrated in Figure 2.

The proper \((w, d)\) configuration of the algorithm is quite delicate. In the full version \([50]\), we analyze how to set \( w \) given a constraint on \( d \) or vice versa. We also show that we achieve the best pruning rate when the matrix size \( d \cdot w \) is minimized (if there is no constraint on \( d \) or \( w \)), thus optimizing the space and pruning simultaneously.

The goal of our algorithm is to ensure that with probability \( 1 - \delta \), where \( \delta \) is an error parameter set by the user, no more than \( w \) TOP \( N \) values are mapped to the same row. In turn, this guarantees that the pruning operation is successful and that all output entries are not pruned. In the following, we assume that \( d \) is given (this can be derived from the amount of per-stage memory available on the switch) and discuss how to set the number of matrix columns. To that end, we use \( w = \left\lfloor \frac{1.3 \ln(d/\delta)}{\ln(\frac{1}{\delta} \ln(d/\delta))} \right\rfloor \) matrix columns. For example, if we wish to find the TOP 1000 with probability 99.99% (and thus, \( \delta = 0.0001 \)) and have \( d = 600 \) rows then we use \( w = 16 \) matrix columns. Having more space (larger \( d \)) reduces \( w \); e.g., with \( d = 8000 \) rows we need just 5 matrix columns. Having too few rows may require an excessive number of matrix columns (e.g., \( w = 288 \) matrix columns are required for \( d = 200 \)) which may be infeasible due to the limited number of pipeline stages. Due to lack of space, the proof of the theorem appears in the full version \([50]\).

**Theorem 2.** Let \( d, N \in \mathbb{N}, \delta > 0 \) such that \( d \geq N \cdot e/\ln \delta^{-1} \) and define \( w = \left\lfloor \frac{1.3 \ln(d/\delta)}{\ln(\frac{1}{\delta} \ln(d/\delta))} \right\rfloor \). Then TOP \( N \) query succeeds with probability at least \( 1 - \delta \).

In the worst case, if the input stream is monotonically increasing, the switch must pass all entries to ensure correctness. In practice, streams are unlikely to be adversarial as the order in which they are stored is optimized for performance. To that end, we analyze the performance on random streams, or equivalently, arbitrary streams that arrive in a random order. Going back to the above example, if we have \( d = 600 \) rows on the switch and aim to find TOP 1000 from a stream of \( m = 8M \) elements, our algorithm is expected to prune at least 99% of the data. For a larger table of \( m = 100M \) entries our bound implies expected pruning of over 99.9%. Observe that the logarithmic dependency on \( m \) in the following theorem implies that our algorithm work better for larger datasets. The following theorem’s proof is deferred to the full version \([50]\).

**Theorem 3.** Consider a random-order stream of \( m \) elements and the TOP \( N \) operation with algorithm parameters \( d, w \) as discussed above. Then our algorithm prunes at least \( m - w \cdot d \cdot \ln(\frac{m-w}{d}) \) of the \( m \) elements in expectation.

**Optimizing the Space and Pruning Rate** The above analysis considers the number of rows \( d \) as given and computes the optimal value for the number of matrix columns \( w \). However, unless one wishes to use the minimal number of matrix columns possible for a given per-stage space constraint, we can simultaneously optimize the space and pruning rate. To that end, observe that the required space for the algorithm is \( \Theta(w \cdot d) \), while the pruning rate is monotonically decreasing in \( w \cdot d \) as shown in Theorem 3. Therefore, by minimizing the product \( w \cdot d \) we optimize the algorithm in both aspects. Next, we note that for a fixed error probability \( \delta \) the value for \( w \) is monotonically decreasing in \( d \) as shown in Theorem 2. Therefore we define \( f(d) = w \cdot d = \frac{d \cdot 1.3 \ln(d/\delta)}{\ln(\frac{1}{\delta} \ln(d/\delta))} \) and minimize it over the possible values of \( d \).

**Example #8: (Probabilistic) DISTINCT Query:** Some DISTINCT queries run on multiple input columns or on variable-width fields that are too wide and exceed the number of bits that can be parsed from a packet. To reduce the bits, we use fingerprints, which are short hashes of all input columns that the query runs on. However, fingerprint collisions may cause the switch to prune entries that have not appeared before and thus provide inaccurate output. Interestingly, not all collisions are harmful. This is because the DISTINCT algorithm hashes each entry into a row in \( \{1, \ldots, d\} \). Thus, a fingerprint collision between two entries is bad only if they are in the same row. We prove the following bound on the required fingerprint length in the full version \([50]\).

\[\text{This omits the flooring of } w \text{ as otherwise the function is not continuous. The actual optimum, which needs to be integral, will be either the minimum } d \text{ for that value or for } w \text{ that is off by 1.}\]

\[\text{Note that for some other queries (e.g., JOIN), fingerprint collisions only affect the pruning rate, but not correctness.}\]
Theorem 4. Denote

\[ M = \begin{cases} 
  e \cdot D/d & \text{if } D > d \ln(2d/\delta) \\
  e \cdot \ln(2d/\delta) & \text{if } d \cdot \ln \delta^{-1}/e \leq D \leq d \ln(2d/\delta), \\
  1.3 \ln(2d/\delta) & \text{if } d \cdot \ln \delta^{-1}/e \leq 0.1 \%
\end{cases} \]

where \( D \) is the number of distinct items in the input. Consider storing fingerprints of size \( f = \lceil \log(d \cdot M^2/\delta) \rceil \) bits. Then with probability \( 1 - \delta \) there are no false positives and the distinct operation terminates successfully.

For example, if \( d = 1000 \) and \( \delta = 0.01\% \), we can support up to \( 500M \) distinct elements using 64-bits fingerprints regardless of the data size. Further, this does not depend on the value of \( w \).

The analysis leverages the balls and bins framework to derive bounds on the sum of square loads, where each load is the number of distinct elements mapped into a row. It then considers the number of distinct elements we can support without having same-row fingerprint collisions. For example, if \( d = 1000 \) and the error probability \( \delta = 0.01\% \), we can support up to \( 500M \) distinct elements using 64-bits fingerprints regardless of the data size. Further, this does not depend on the value of \( w \). We also provide a rigorous analysis of the pruning rate in Additionally, we analyze the expected pruning rate in random-order streams and show, in the full version \([50]\), that we can prune at least an \( \Omega \left( \frac{w \cdot d}{D} \right) \) fraction of the entries, where \( D \) is the number of distinct elements in the input.

6 Handling multiple queries

Cheetah supports the use case where the query is not known beforehand but only the set of queries (e.g., DISTINCT, TOP N, and JOIN) we wish to accelerate. Alternatively, the workload may contain complex queries that combine several of our operations. In this scenario, one alternative would be to reprogram the switch once a query arrives. However, this could take upwards of a minute and may not be better than to perform the query without Cheetah. Instead, we concurrently pack the different queries that we wish to support on the switch data plane, splitting the ALU/memory resources between these. This limits the set of queries we can accommodate in parallel, but allow for interactive query processing in a matter of seconds and without recomipling the switch. Further, not all algorithms are heavy in the same type of resources. Some of our queries (e.g., SKYLINE) require many stages but few ALUs and only a little SRAM. In contrast, JOIN may use only a couple of stages while requiring most of the SRAM in them. These differences enable Cheetah to efficiently pack algorithms on the same stages.

At the switch, all queries will be performed on the incoming data giving us a prune/no-prune bit for each query. Then we have a single pipeline stage that selects the bit relevant to the current query. We fit multiple queries by repurposing the functionality of ALU results and stages. We evaluate one such combined query in figure 5. Query A is a filtering query and query B is a SUM + group by query. To prune the filtering query, we only use a single ALU and 32 bits of stage memory (1 index of a 32 bit register) in a stage. We use the remaining ALUs in the same stage to compute 1) hash values and 2) sums required for query B as discussed in our pruning algorithms. We also use the remaining stage memory in that same stage to store SUM results ensuring the additional filter query has no impact on the performance of our group by query.

In more extreme examples, where the number of computation operations required exceeds the ALU count on the switch, it is still possible to fit a set of queries by reusing ALUs and registers for queries with similar processing. As an example, an ALU that does comparisons for filtering queries can be reconfigured using control plane rules to work as part of the comparator of a TOP N or HAVING query. We can also use a single stage for more than one task by partitioning its memory e.g. dedicating part of to fingerprinting for DISTINCT and another part to store SKYLINE prune points.

7 Implementation

7.1 Cheetah prototype

We built the Cheetah dataplane along with in-network pruning using a Barefoot Tofino \([39]\) switch and P4 \([13]\). Each query requires between 10 to 20 control plane rules excluding the rules needed for TCP/IP routing and forwarding. Any of the Big Data benchmark workloads can be configured using less than 100 control plane rules. We also developed a DPDK-based Cheetah end-host service using about 3500 lines of C.

We deploy five Spark workers along with an instance of CWorker connected to the switch via DPDK-compliant 40G Mellanox NICs. We restrict the NIC bandwidth to 10G and 20G for our evaluation. All workers have two CPU cores and 4 GB of memory. The CWorker sends data to the master via UDP at a rate of \( \sim 10 \) million packets per second (i.e. a throughput of \( \sim 5.1 \) Gbps since the minimum ethernet frame is 64 bytes) with one entry per packet. We use optimized tasks for Spark for a fair comparison. We also mount a linux tmpfs RAM disk on workers to store the dataset partitions allowing Spark to take advantage of its main-memory optimized query plans.

Spark optimizes the completion time by minimizing data movement. In addition to running tasks on workers to reduce the volume sent to the master, Spark compresses the data and packs multiple entries in each packet (often, the maximum allowed by the network MTU). In contrast, Cheetah must send the data uncompressed while packing only a small number of entries in each packet. Spark also leverages an optimization called *late materialization* \([3]\) in which only
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Defaults</th>
<th>#stages</th>
<th>#ALUs</th>
<th>SRAM</th>
<th>#TCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTINCT</td>
<td>FIFO*, LRU</td>
<td>$w = 2, d = 4096$</td>
<td>$[w/A]$</td>
<td>$w$</td>
<td>$(d \cdot w) \times 64b$</td>
</tr>
<tr>
<td>SKYLINE</td>
<td>SUM, APH</td>
<td>$D = 2, w = 10$</td>
<td>$[\log_2 D + 2w \cdot (w+1)] 2^{\lceil \log_2 D \rceil - 1 + w(D+1)}$</td>
<td>$w(D+1) \times 64b$</td>
<td>0</td>
</tr>
<tr>
<td>TOP N</td>
<td>Det, Rand</td>
<td>$N = 250, w = 4$</td>
<td>$w + 1$</td>
<td>$w + 1$</td>
<td>$(w + 1) \times 64b$</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>BF*, RBF</td>
<td>$M = 4MB, H = 3$</td>
<td>$[d/A]$</td>
<td>$d$</td>
<td>$(d \cdot w) \times 64b$</td>
</tr>
</tbody>
</table>

Table 2: Summary of the resource consumption of our algorithms. Here, $A$ is the number of ALUs per stage on the switch. The algorithms denoted by (*) assume that same-stage ALUs can access the same memory space. For SKYLINE the above assumes that the dimension satisfies $D \leq A$.

Figure 3: The control flow of Apache Spark and Cheetah

a metadata stream is sent in the first stage, and the entire entries are requested by the master once it computes which tuples are part of the result. We expand on how Cheetah supports late materialization in the full version [50].

Currently, our prototype includes the DISTINCT, SKYLINE, TOP N, GROUP BY, JOIN, and filtering queries. We also support combining these queries and running them in parallel without reprogramming the switch.

Cheetah Modules We create two background services that communicate with PySpark called Cheetah Master (CMaster) and Cheetah Worker (CWorker) running on the same servers that run the Spark Master and Spark Workers respectively. The CMaster bypasses a supported PySpark query and instead sends a control message to all CWorkers providing the dataset and the columns relevant to the query. The CWorker packs the necessary columns into UDP packets with Cheetah headers and sends them via Intel DPDK [25]. Our experiments show a CWorker can generate over $\approx 12$ million packets per second when the link and NIC are not a bottleneck.

In parallel, the master server also communicates with the switch to install the control plane rules relevant to the query. The switch parses and prunes some of the packets it processes. The remaining packets are received at the master using an Intel DPDK packet memory buffer, are parsed, and copied into userspace. The remaining processing for the query is implemented in C. The Cheetah master continuously processing all entries it receives from the switch until receiving FINs from all CWorkers, indicating that data transmission is done. Finally, the CMaster sends the final set of row ids and values to the Python PySpark script (or shell).

Switch logic We use Barefoot Tofino [39] and implement the pruning algorithms (see §3) in the P4 language. The switch parses the header and extracts the values which then proceed to the algorithm processing. The switch then decides if to prune each packet or forward it to the master. It also participates in our reliability protocol, which takes two pipeline stages on the hardware switch.

Resource Overheads All our algorithms are parametric and can be configured for a wide range of resources. We summarize the hardware resource requirements of the algorithms in Table 2 and expand on how this is calculated in the full version of the paper [50].

7.2 Communication protocol

Query and response format: For communication between the CMaster node and CWorkers (passing through and getting pruned by the switch), we implement a reliable transmission protocol built on top of UDP. Each message contains the entry identifier along with the relevant column values or hashes. Our protocol runs on a separate port from Spark and uses a separate header. It also does not use Spark’s
serialVersionUID

8.1 Evaluation setup

Benchmarks: Our test-bed experiments use the Big Data [4] and TPC-H [2] benchmarks. From the Big Data benchmark, we run queries A (filtering), B (we offload group-by), and A+B (both A and B executed sequentially).

For TPC-H, we run query 3 which consists of two join operations, three filtering operations, a group-by, and a top N. We also evaluate each algorithm separately using a dedicated query on the Big Data benchmark’s tables. All queries are shown in the full version of the paper [50].

8.2 Testbed experiments

We run the BigData benchmark on a local cluster with five workers and one master, all of which are connected directly to the switch. Our sample contains 31.7 million rows for the uservisits table and 18 million rows for the rankings table. We run the TPC-H benchmark at its default scale with one worker and one master. Cheetah offloads the join part of the TPC-H because it takes 67% of the query time and is the most effective use of switch resources.

8.2.1 Benchmark Performance

Figure 5 shows that Cheetah decreases the completion time by 64 – 75% in both BigData B, BigData A+B, and TPC-H Query 3 compared to Spark’s 1st run and 47 – 58% compared to subsequent runs. Spark’s subsequent runs are faster than 1st run because Spark indexes the dataset based on the given workload after the 1st run. Cheetah reduces the completion time by 40 – 72% for other database operations such as distinct, groupby, Skyline, TopN, and Join. Cheetah improves performance on these computation intensive aggregation queries because it reduces the expensive task computation Spark runs at the workers by offloading it to the switch’s data plane instead.

BigData A (filtering) does not have a high computation overhead. Hence Cheetah has performance comparable to

8.3 Case studies

We demonstrate the benefits of Cheetah in three case studies: 1) Joins, 2) Computations, and 3) Aggregations.

1) Joins: Our test-bed experiments show that Cheetah has 40 – 200% improvement in query completion time over Spark. Our simulations show that Cheetah achieves a high pruning ratio with a modest amount of switch resources.
Figure 5: A comparison of Cheetah and Spark in terms of completion time on the Big Data benchmark for the benchmark queries (first four) and the other queries supported by Cheetah.

Figure 6: The performance of Cheetah vs. Spark SQL on DISTINCT query.

Spark’s 1st run but worse than Spark’s subsequent runs. This is because Cheetah has the extra overhead of serializing data at the workers to allow processing at the switches. This serialization adds more latency than the time saved by switch pruning. Cheetah performs the combined query A + B faster than the sum of individual completion times. This is because it pipelines the pre-processing of columns for the combined query resulting in faster serialization at CWorker.

8.2.2 Effect of Data Scale and Number of Workers In Figure 6a, we vary the number of entries per partition (worker) while keeping the total number of entries fixed. Not only is Cheetah quicker than Spark, the gap widens as the data scale grows. Therefore Cheetah may also offer better scalability for large datasets. Figure 6b shows the performance when fixing the overall amount of entries and varying the number of workers. Cheetah improves Spark by about the same factor with a different number of partitions. In both these experiments, we ignore the completion time of Spark’s first run on the query and only show subsequent runs (which perform better due to caching / indexing and JIT compilation effects [36, 42]).

8.2.3 Effect of Network Rate Unlike Spark, which is often bottlenecked by computation [42], Cheetah is mainly limited by the network when using a 10G NIC limit. We run Cheetah using a 20G NIC limit and show a breakdown analysis of where each system spends more time. Figure 8 illustrates how Cheetah diminishes the time spent at the worker at the expense of more sending time and longer processing at the master. The computation here for Cheetah is done entirely at the master server, with the workers just serializing packets to send them over the network. When the speed is increased to 20G, the completion time of Cheetah improves by nearly 2x, meaning that the network is the bottleneck. Similarly to Section 8.2.2, we discard Spark’s first run.

8.2.4 Comparison with NetAccel [32] NetAccel is a recent system that offloads entire queries to switches. Since the switch data plane limitations may not allow a complete computation of queries, NetAccel overflows some of the data to the switch’s CPU. At the end of the execution, NetAccel drains the output (which is on the switch) and provides it to the user. Cheetah’s fundamental difference is that it does not aim to complete the execution on the switch and only prunes to reduce data size. As a result, Cheetah is not limited to queries whose execution can be done on the switch (NetAccel only supports join and group by), does not need to read the output from the switch (thereby saving latency), and can efficiently pipeline query execution.

As NetAccel stores results at switches, it must drain the output to complete the execution. This process adds latency, as shown in figure 7. We note that NetAccel’s code is not publically available, and the depicted results are a lower bound obtained by measuring the time it takes to read the output from the switch. That is, this lower bound represents
We use simulations to study the pruning rates under various algorithm settings and switch constraints. The pruning rates dictate the number of entries that reach the master and therefore impact the completion time.

To understand how pruning rate affects completion time, we measure the time the master needs to complete the execution once it gets all entries. Figure 9 shows that the time it takes the master to complete the query significantly increases with more unpruned entries. The increase is super-linear in the unpruned rate since the master can handle each arriving entry immediately when almost all entries are pruned. In contrast, when the pruning rate is low, the entries buffer up at the master, causing an increase in the completion time.

The desired pruning rate depends on the complexity of a query’s software algorithm. For example, TOP N is implemented on the master using an \( N \)-sized heap and processes millions of entries per second. In contrast, SKYLINE is computationally expensive and thus we should prune more entries to avoid having the master become a bottleneck. We only show three queries here. Experiments for remaining queries appear in the full version [50].

**8.3 Pruning Rate Simulations**

We use simulations to study the pruning rates under various network rates. Spark's bottleneck is not the network and it does not improve when using a faster NIC.

![Figure 8: breakdown of Spark and Cheetah’s delay for different network rates.](image)

**Figure 8:** breakdown of Spark and Cheetah’s delay for different network rates. Spark’s bottleneck is not the network and it does not improve when using a faster NIC.

![Figure 9: The time it takes the Master to complete DISTINCT and max-GROUP BY queries for a given pruning rate.](image)

**Figure 9:** The time it takes the Master to complete DISTINCT and max-GROUP BY queries for a given pruning rate.

To understand how pruning rate affects completion time, we measure the time the master needs to complete the execution once it gets all entries. Figure 9 shows that the time it takes the master to complete the query significantly increases with more unpruned entries. The increase is super-linear in the unpruned rate since the master can handle each arriving entry immediately when almost all entries are pruned. In contrast, when the pruning rate is low, the entries buffer up at the master, causing an increase in the completion time.

The desired pruning rate depends on the complexity of a query’s software algorithm. For example, TOP N is implemented on the master using an \( N \)-sized heap and processes millions of entries per second. In contrast, SKYLINE is computationally expensive and thus we should prune more entries to avoid having the master become a bottleneck. We only show three queries here. Experiments for remaining queries appear in the full version [50].

**Pruning Rate vs. Resources Tradeoff** We evaluate the pruning rate that Cheetah achieves for given hardware constraints. In all figures, OPT depicts a hypothetical stream algorithm with no resource constraints. For example, in a TOP N it shows the fraction of entries that were among the \( N \) largest entries from the beginning of the stream. Therefore, OPT is an upper bound on the pruning rate of any switch algorithm. The results are depicted in Figure 10a-10c. We ran each randomized algorithm five times and used two-tailed Student t-test to determine the 95% confidence intervals. We configured the randomized algorithms to \( \geq 99.99\% \) success probability. In 10a we see that using \( w = 2 \) and \( d = 4096 \) Cheetah can prune all non-distinct entries; with smaller \( d \) or the FIFO policy the pruning rate is slightly lower but Cheetah still prunes over 99% of the entries using just a few KB of space. In 10b we see SKYLINE results; as expected, for the same number of points, APH outperforms the SUM heuristic and prunes all non-skyline points with \( w = 20 \). Both APH and SUM prune over 99% of the entries with \( w \leq 7 \) while Baseline, which stands for an algorithm that store \( w \) arbitrary points for pruning, requires \( w \geq 20 \) for 99% pruning. Both heuristics allow the switch to “learn” a good set of points to use for the pruning. 10c shows TOP N and illustrates the power of the randomized approach. While the deterministic algorithm can run with fewer stages and ensure correctness, if we allow just 0.01% chance of failure, we can significantly increase the pruning rate. Here, the randomized algorithm reaches about 99.995% pruning, leaving about 5 times the optimal number of packets. The strict requirement for high-success probability forces the algorithm to only prune entries which are almost certainly not among the top N.

**Pruning Rate vs. the Data Scale:** The pruning rate of different algorithms behaves differently when the data scale grows. Here, each data point refers to the first entries in the relevant data set. Figures 10d - 10f show how the pruning rate varies as the scale grows. For the shown queries, Cheetah achieves better pruning rate for larger data. For DISTINCT it is because we cannot prune the first occurrence of an output key, but once our data structure has these reflected it gets better pruning. In SKYLINE and TOP N, a smaller fraction of input entries are needed for the output as the data scale grows, allowing the pruning of more entries. In contrast, the algorithms for JOIN and HAVING have better pruning rates for smaller data sets, as shown in the full version [50]. In JOIN, the algorithm experiences more false positives as the data keep on coming and therefore prunes a smaller fraction of the entries. The HAVING query is conceptually different; as it asks for the codes for languages whose sum-of-ad-revenue is larger than $1M$, the output is empty if the data is too small. The one-sided error of the Count Min sketch
that we use guarantees that we do not miss any of the correct output keys but the number of false positives increases as the data grows. Nevertheless, with as few as 512 counters for each of the three rows, Cheetah gets near-perfect pruning throughout the evaluation.

9 Extensions

Multiple switches: We have considered a single programmable switch in the path between the workers and the master. However, having multiple switches boosts our performance further. For example, we can use a "master switch" to partition the data and offload each partition to a different switch. Each switch can perform local pruning of its partition and return it to the master switch which prunes the data further. This increases the hardware resources at our disposal and allows superior pruning results.

DAG of workers: Our paper focuses on the case where there is one master and multiple workers. However, in large scale deployments or complex workloads, query planning may result in a directed acyclic graph (DAG) of workers, each takes several inputs, runs a task, and outputs to a worker on the next level. In such cases, we can run Cheetah at each edge in which data is sent between workers. To distinguish between edges, each has a dedicated port number and a set of resources (stages, ALUs, etc.) allocated to it. To that end, we use the same packing algorithm described in §6.

Packing multiple entries per packet: Cheetah spends a significant portion of its query processing time on transmitting the entries from the workers. This is due to two factors: first, it does not run tasks on the workers that filters many of the entries; second, it only packs one entry in each packet. While the switch cannot process a very large number of entries per packet on the switch due to limited ALUs, we can still pack several (e.g. four) entries in a packet thereby significantly reducing this delay. P4 switches allow popping header fields [13] and thereby support pruning of some of the entries in a packet. The limit on the number of entries in each packet depends on the number of ALUs per stage (all our algorithms use at least one ALU per entry per stage), the number of stages (we can split logical stage to several if the pipeline is long enough). Our DISTINCT, TOP N, and GROUP BY algorithms support multiple entries per packet while maintaining correctness: if several entries are mapped to the same matrix row, we can avoid processing them while not pruning the entries.
10 Related Work

This work has not been published elsewhere except for a 2-page poster at SIGCOMM [51]. The poster discusses simple filtering, DISTINCT, and TOP N. This work significantly advances the poster by providing pruning algorithms for 4 additional queries, an evaluation on two popular benchmarks, and a comparison with NetAccel [32]. This work also discusses probabilistic pruning, optimizing multiple queries, using multiple switches, and a reliability protocol.

**Hardware-based query accelerators:** Cheetah follows a trend of accelerating database computations by offloading computation to hardware. Industrial systems [1, 41] offload parts of the computation to the storage engine. Academic works suggest offloading to FPGAs [5, 15, 47, 53], SSDs [17], and GPUs [19, 45, 48]. These either consider offloading the entire operation to hardware [32, 53], or doing a per-partition exact aggregation/filtering before transmitting the data for aggregation [17]. However, exact query computation on hardware is challenging and these only support basic operations (e.g., filtering [1, 17]) or primitives (e.g., partitioning [31] or aggregation [16, 53]).

Cheetah uses programmable switches which are either cheaper or have better performance than alternative hardware such as FPGAs. Compared to FPGAs, switches handle two orders of magnitude more throughput per Watt [52] and ten times more throughput per dollar [26] [52]. GPUs consume 2-3x more energy than FPGAs for equivalent data processing workloads [43] and double the cost of FPGA with similar characteristics [43]. A summary of the attributes of the different alternatives appears in Table 3. Switches are also readily available in the networks at no additional cost. We offload partial functions on switches using the pruning abstraction and support a variety of database queries. Sometimes, FPGAs and GPUs also incur extra data transmission overhead. For example, GPU’s separate memory system also introduces significant performance overhead with extra computation and memory demand [53]. When an FPGA is attached to the PCIe bus [1, 5], we have to copy the data to and from the FPGA explicitly [53]. One work has used FPGAs as an in-datapath accelerator to avoid this transfer [53].

Moreover, switches can see the aggregated traffic across workers and the master, and thus allow optimizations across data partitions. In contrast, FPGAs are typically connected to individual workers due to bandwidth constraints [1] and can only optimize the query for each partition. That said, Cheetah complements these works as switches can be used with FPGA and GPU-based solutions for additional performance.

### Table 3: Performance comparison of hardware choices.

<table>
<thead>
<tr>
<th>System</th>
<th>Throughput</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>10-100Gbps</td>
<td>10-100µs</td>
</tr>
<tr>
<td>FPGA [53]</td>
<td>10-100µps</td>
<td>5-10µs</td>
</tr>
<tr>
<td>SmartNIC [34]</td>
<td>5-10µs</td>
<td>12.5µps</td>
</tr>
</tbody>
</table>

**Offloading to programmable switches:** Several works use programmable switches for offloading different functionality that was handled in software [12, 14, 21, 27–30, 33, 35, 46]. Cheetah offloads database queries, which brings new challenges to fit the constrained switch programming model because database queries often provide a large amount of data and require diverse computations across many entries. One opportunity in databases is that the master can complete the query from the preloaded data set.

In the network telemetry context, researchers proposed Sonata, a general monitoring abstraction that allows scripting for analytics and security applications [20]. Sonata supports filtering, map and a constrained version of distinct in the data plane but relies on a software stream processor for other operations (e.g., GROUP BY, TOP N, SKYLINE). Conceptually, as Sonata offloads only operations that can be fully computed in the data plane, its scope is limited. Sparser [44] accelerates text-based filtering using SIMD.

Recently, NetAccel [32] suggested using programmable switches for query acceleration. We discuss and evaluate the differences between Cheetah and NetAccel in §8.2.4. Jumpgate [37] also suggests accelerating Spark using switches. It uses a method similar to NetAccel. However, while Cheetah and NetAccel are deployed in between the worker and master server, Jumpgate stands between the storage engine and compute nodes. Jumpgate does not include an implementation, is specific to filtering and partial aggregation, and cannot cope with packet loss.

11 Conclusion

We present Cheetah, a new query processing system that significantly reduces query completion time compared to the current state-of-the-art for a variety of query types. Cheetah accelerates queries by leveraging programmable switches while using a pruning abstraction to fit in-switch constraints without affecting query results.

12 Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We thank Mohammad Alizadeh and Geeticka Chauhan for their help and guidance in the early stages of this project, and Andrew Huang for helping us validate Spark SQL queries. This work is supported by the National Science Foundation under grant CNS-1829349 and the Zuckerman foundation.

### References


Approaches to Large-Scale Data Analysis. ACM SIGMOD, 2009.