Partitioning & Clustering

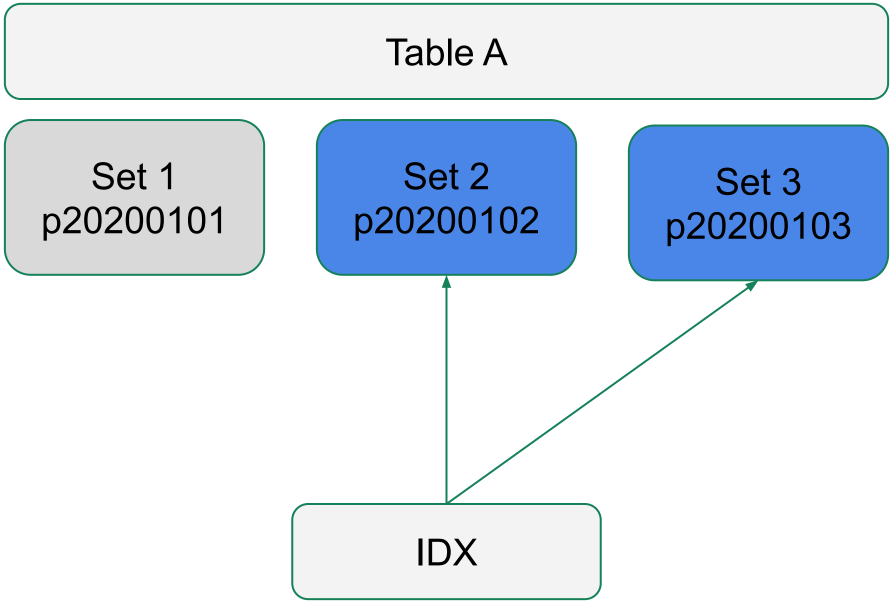
Google Cloud

Partitioning and Clustering

- User provided directives that influence the layout of data in a table.

```
CREATE TABLE T (eventDate TIMESTAMP,
                customerId INTEGER,
                itemId STRING,
                ...,
                ...);
PARTITION BY DATE(eventDate)
CLUSTER BY customerId, itemId;
```

Partitioning



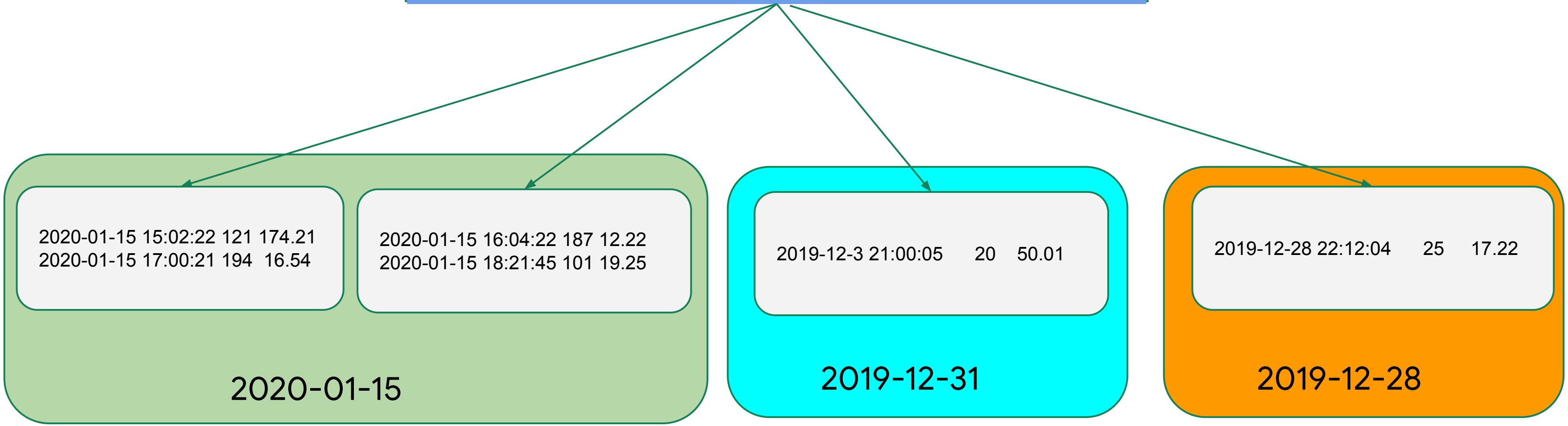
SELECT … WHERE eventDate >= “20200102”

Partitioning

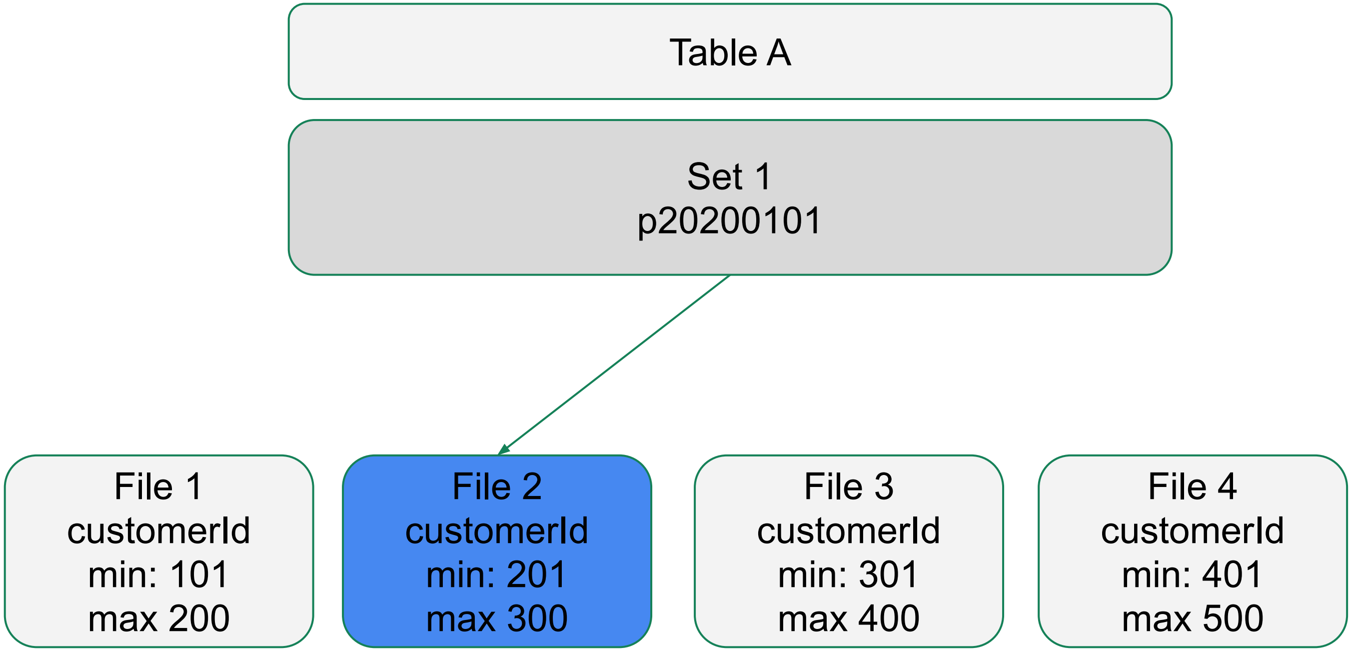
- Data automatically partitioned at write time.
- Each partition behaves like its own table.
- Metadata maintained for each partition.
- Provides strict guarantees for bytes scanned and billed.
- Query cost known upfront.

Writing to a partitioned table

eventDate	customerId	value
2019-12-31 21:00:05	20	50.01
2019-12-28 22:12:04	25	17.22
2020-01-15 17:00:21	194	16.54
2020-01-15 16:04:22	187	12.22
2020-01-15 18:21:45	101	19.25
2020-01-15 15:02:22	121	174.21

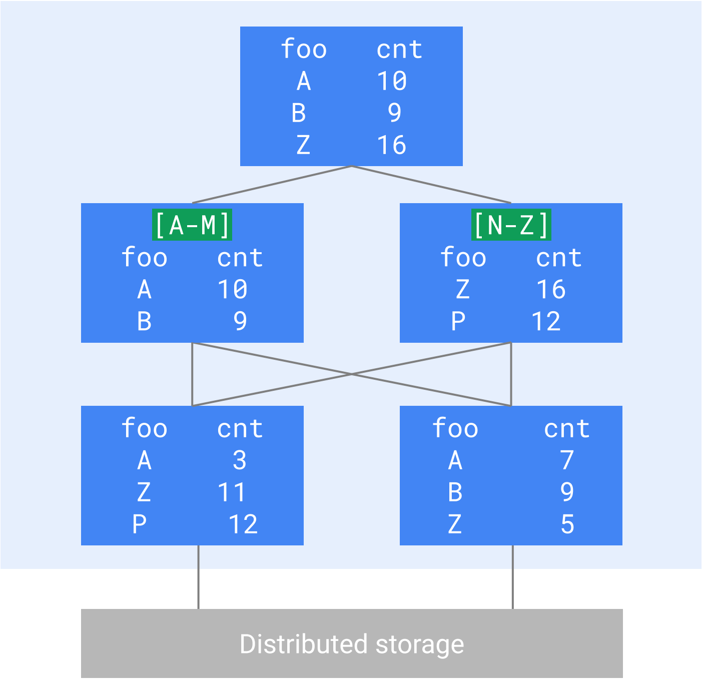


Clustering



SELECT … WHERE customerId = 275

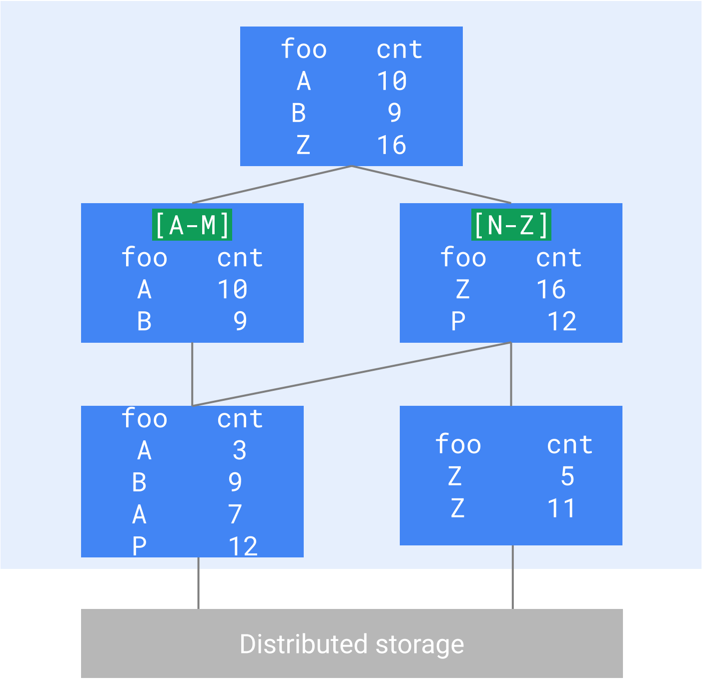
Clustering for aggregation




```
SELECT foo, COUNT(*) as cnt
FROM `...`
GROUP BY 1
```

Unclustered data

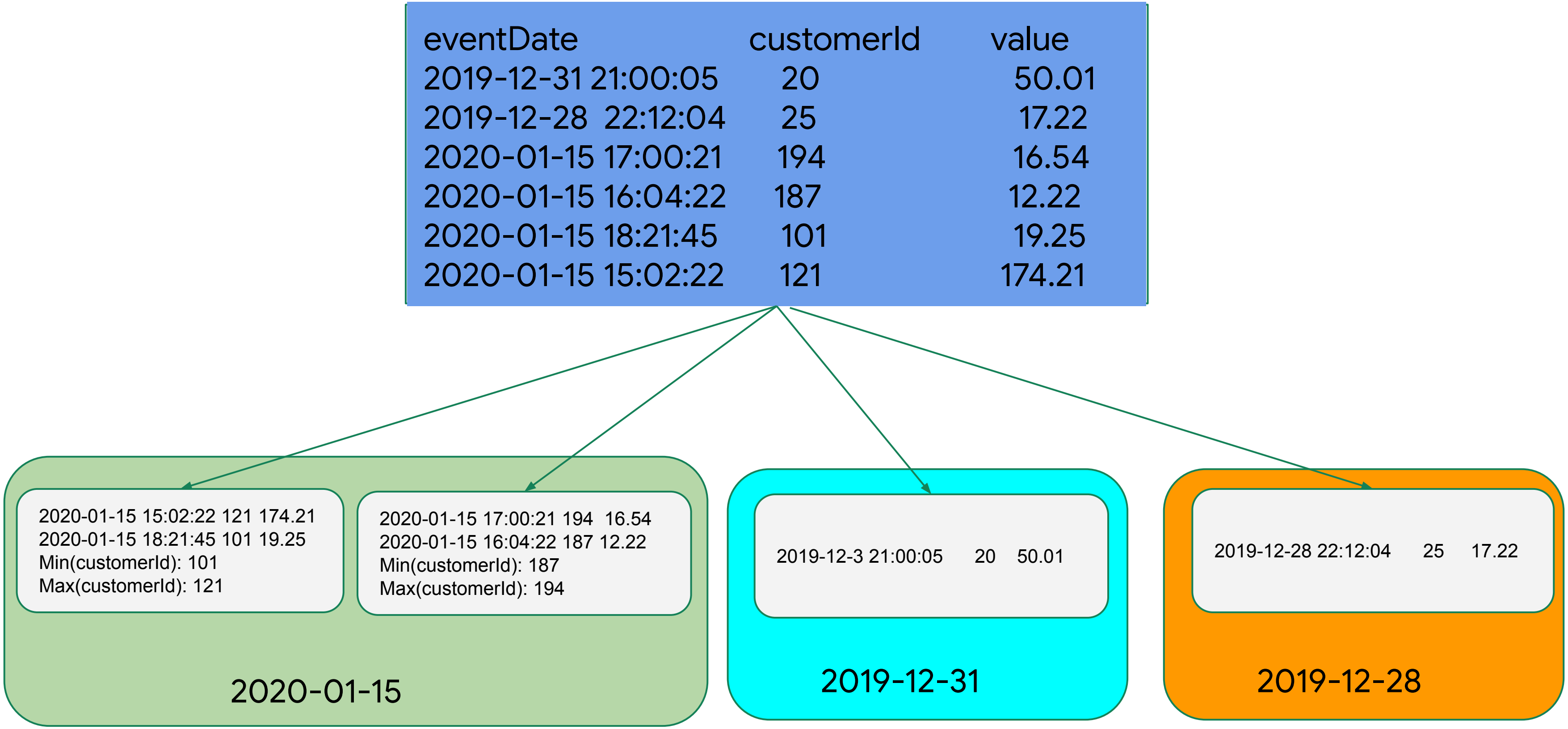
Clustering for aggregation



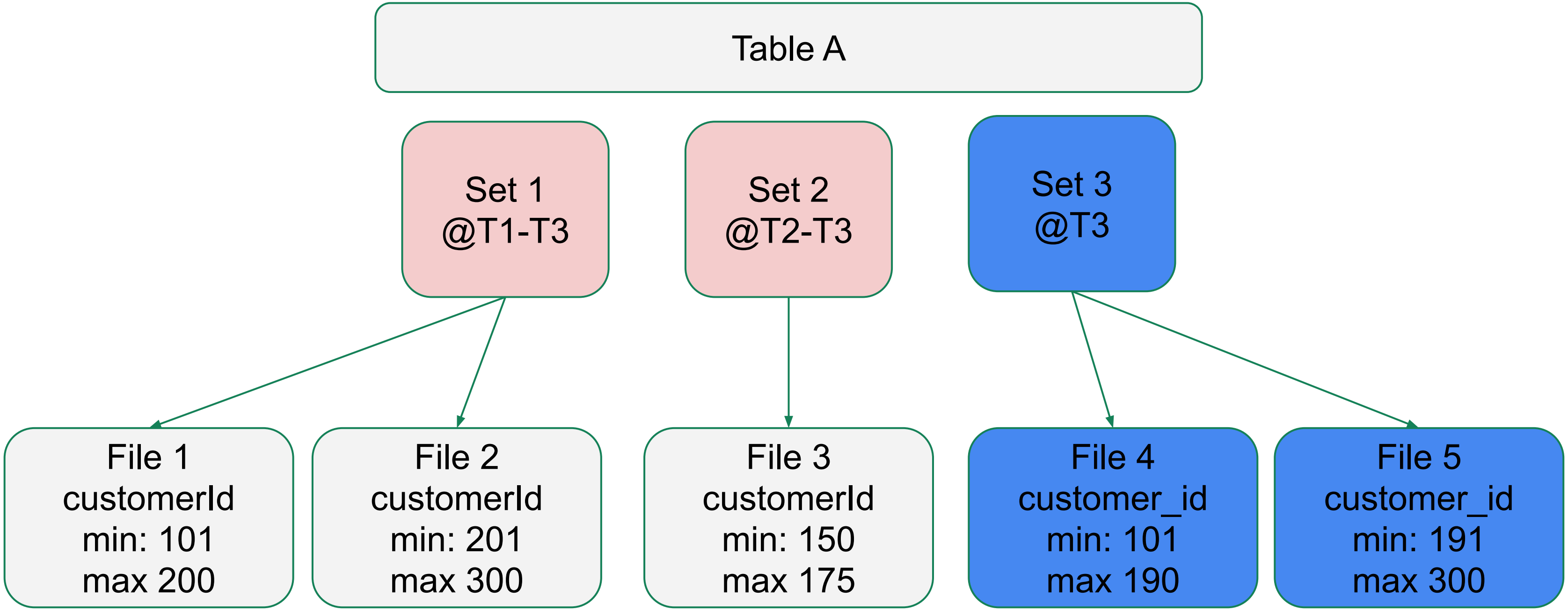

```sql
SELECT foo, COUNT(*) as cnt
FROM `...`
GROUP BY 1
```

Data is clustered by column foo

Writing to a partitioned and clustered table



Reclustering



Google Cloud Platform

BigQuery: Distributed query execution and data shuffling

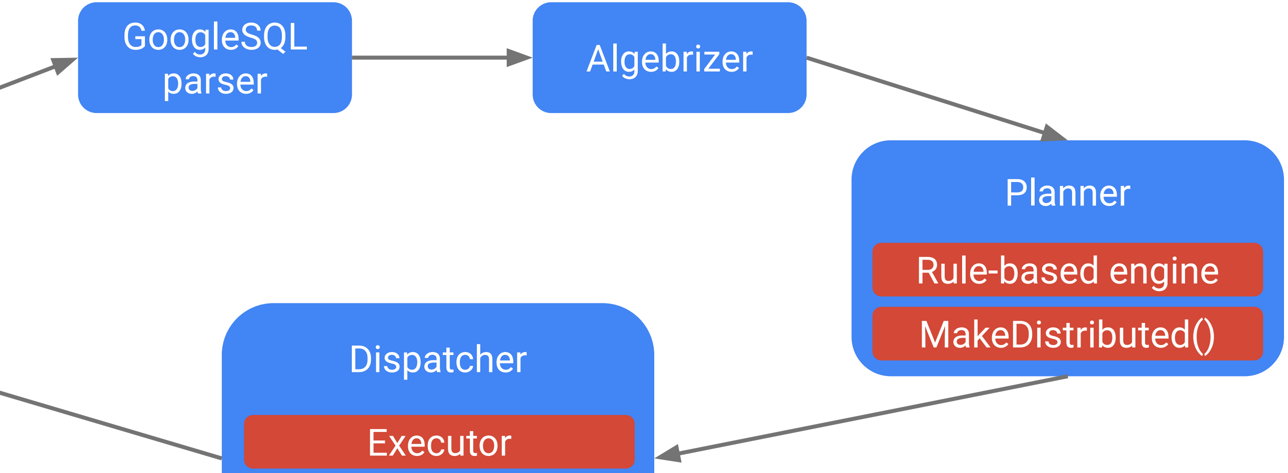
Distributed query execution and data shuffling

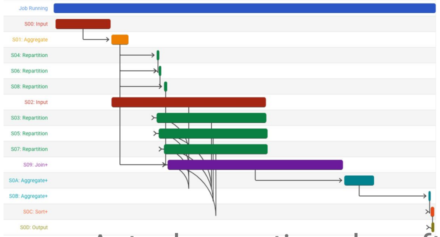
Query execution in a nutshell

Data shuffling

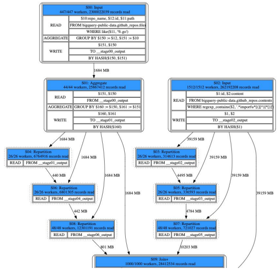
Distributed query execution

Query execution pipeline



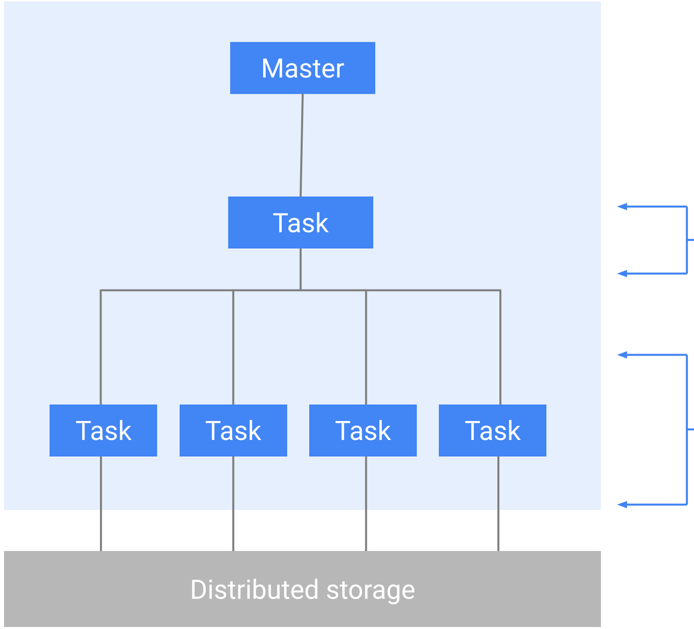


Actual execution plan after dynamically chosen algorithms and resulting degree of parallelism



Original execution plan with default choice of algorithms and best-estimate degree of parallelism

Simple query example




```
SELECT COUNT(*) FROM
`bigquery-public-data.samples.wikipedia`
WHERE title LIKE "S%o%i%y"
```

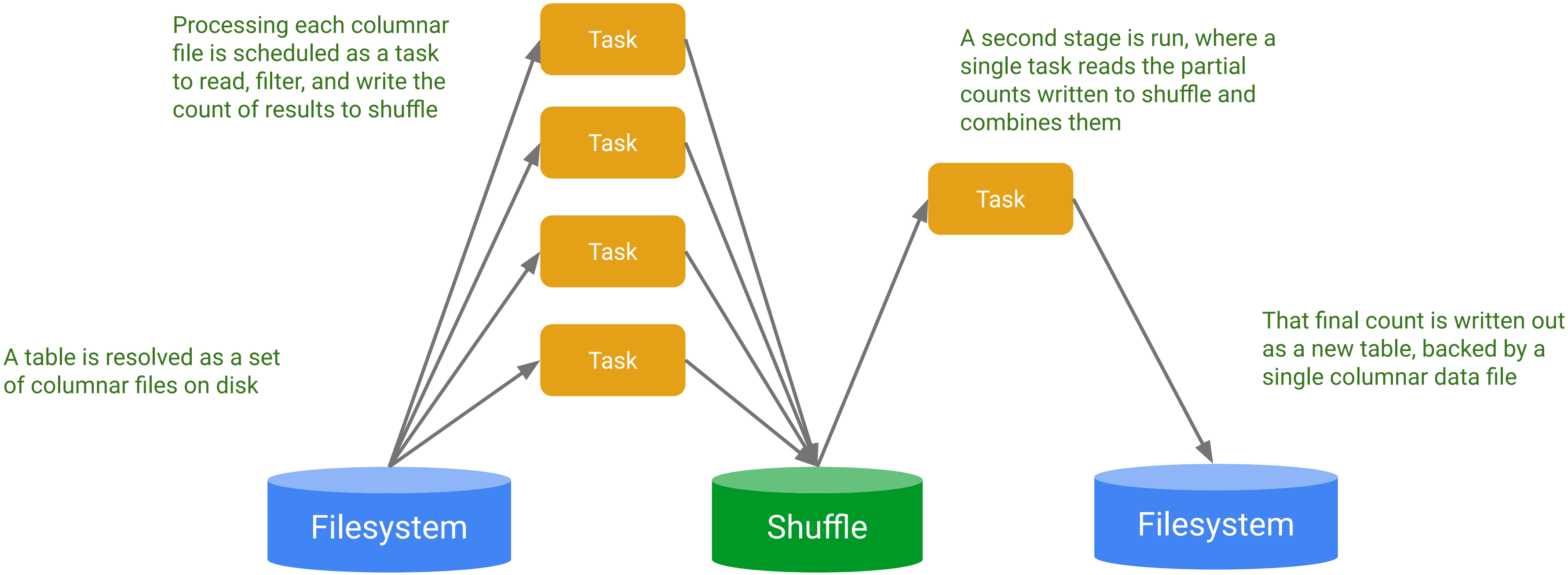
Stage 2: Sum

READ `$20 FROM __stage01_output`
AGGREGATE `$10 := SUM_OF_COUNTS($20)`
WRITE `$10 TO __stage01_output`

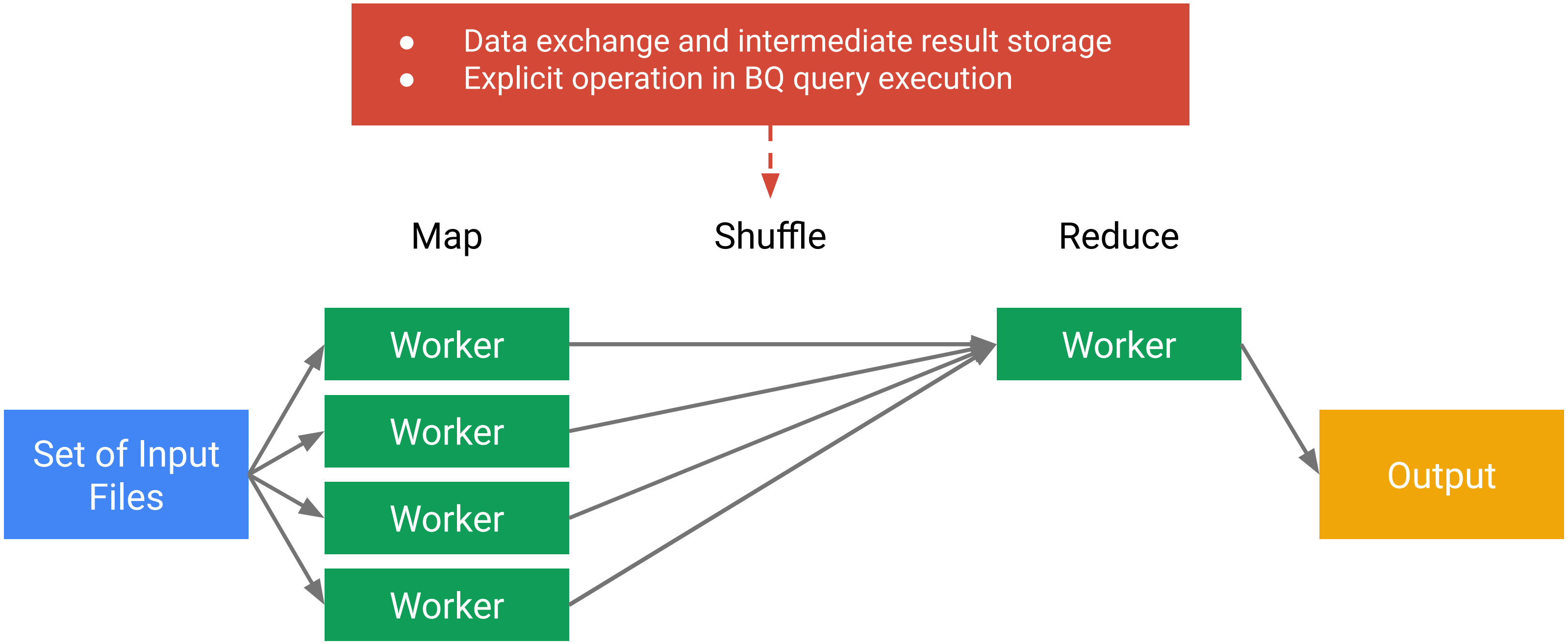
Stage 1: Filter, Count

READ `$1:title FROM`
`bigquery-public-data.samples.wikipedia`
`WHERE like($1, 'S%o%i%y')`
AGGREGATE `$20 := COUNT_STAR()`
WRITE `$20 TO __stage01_output`

With more detail



Parallel to Hadoop/MapReduce



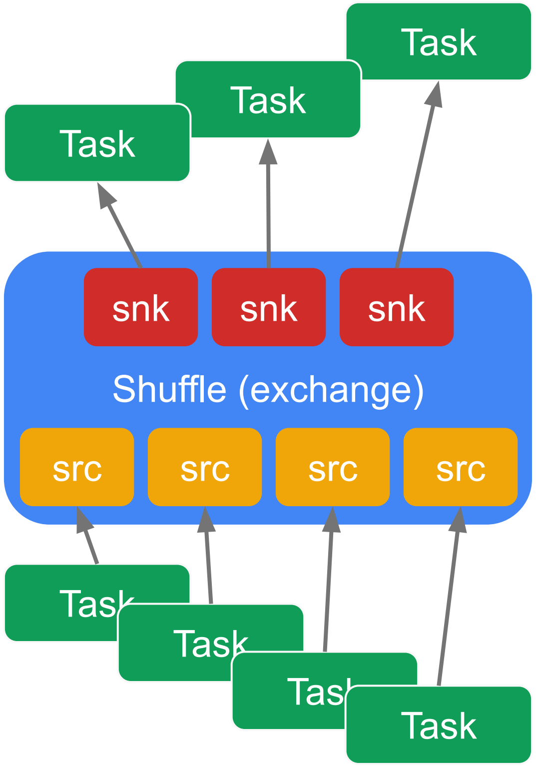
Parallel to traditional parallel query execution

Exchange operator

Pluggable way of changing degree of parallelism in query execution

- M inputs, N (disjoint) outputs
  - Read data from multiple, say M inputs
  - Figure out receiving output through some partitioning scheme (e.g., hash, range)
  - Write data to N outputs

- The shuffle is the BigQuery-specific implementation of an exchange
  - *Sources* model the exchange inputs
  - *Sinks* model the exchange outputs
  - Data reads and writes are orchestrated through Mindmeld, a distributed main memory file system



Questions?

BigQuery distributed execution overview

Query stage: a set of processing tasks corresponding to a query fragment surrounded by distributed operators

Distributed operators: SHUFFLE, DISTRIBUTED UNION, BROADCAST

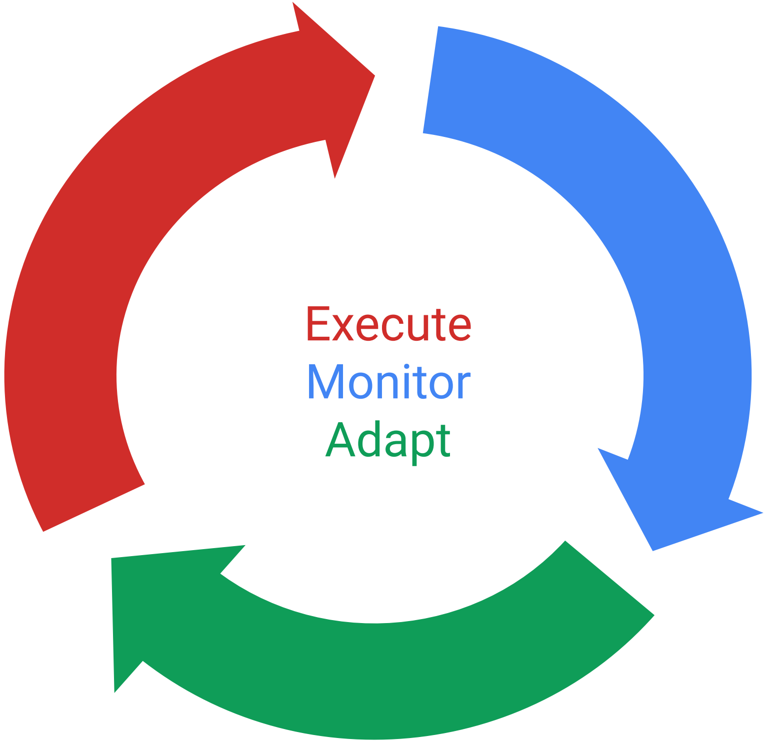
Some distributed operators added during query planning, some added dynamically during execution

Query stages are executed on shards and results are written to Shuffle

Operators using Shuffle:

- JOIN
- AGGREGATION
- PARTITION BY
- MATERIALIZE/EXPORT DATA
- ORDER BY (distributed sort)

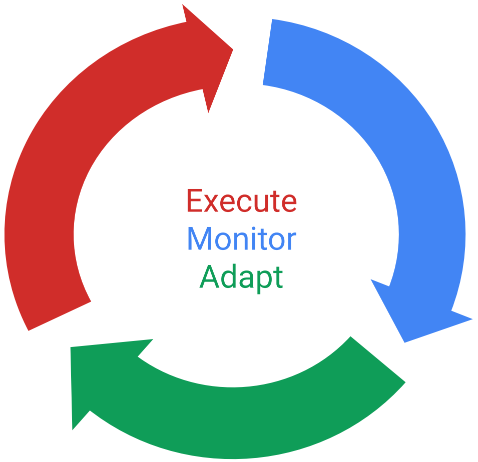
Dynamic query execution overview



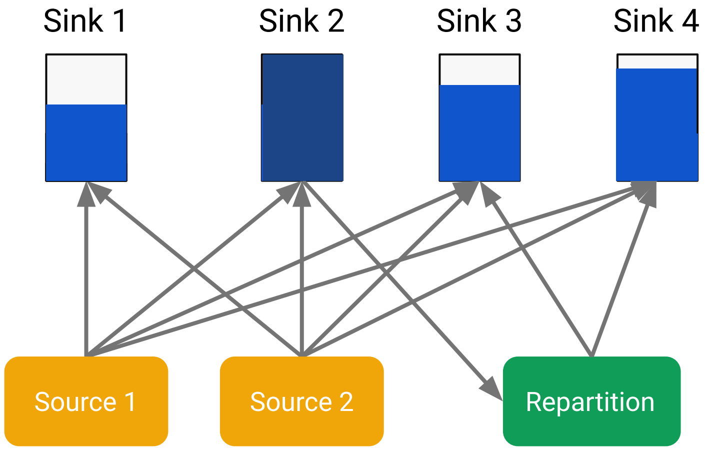
- Execute query stages bottom up
- Collect and monitor various statistics
- Create and cancel query stages
- Adapt query stage plans
- Decide parallelism level
- Mechanisms
    - Dynamic partitioning
    - Plan adaptation

Dynamic partitioning

- Shuffle data starting with an initial number of sinks
- Determine sink target size based on memory usage estimate of the consuming operator or target file data size
- Shuffle Monitor monitors sink sizes. If a sink is over limit:
    - Ask sources to write to new sinks
    - Repartition data from old sink to the new sinks
- Supports partitioning by HASH, RAND, RANGE



Dynamic partitioning example



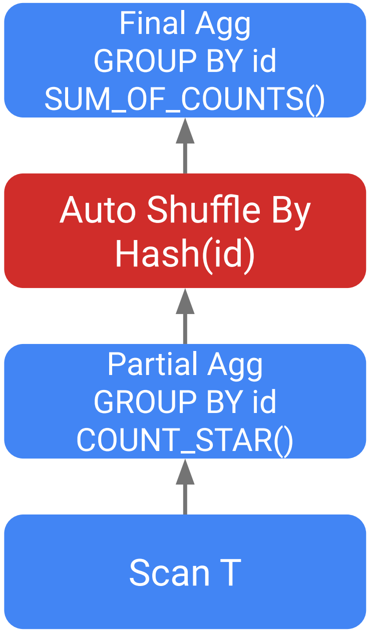
- Sources start writing to Sink 1 and 2
- Shuffle Monitor detects Sink 2 is over limit
- Partitioning scheme changed and sources stop writing to Sink 2 and start writing to Sink 3 and 4
- Repartitioning stage scheduled to repartition Sink 2 into Sink 3 and 4
- Repartitioning Sink 2 is done and no reads/writes from/to it happen

Dynamic partitioning - GROUP BY

```
SELECT COUNT(*) FROM T GROUP BY id
```

- Query planner
  - Splits aggregation into local and global aggregation
  - Places a shuffle operator between them

- Shuffle operator
  - Uses dynamic partitioning to partition data
  - Based on a per-partition memory budget

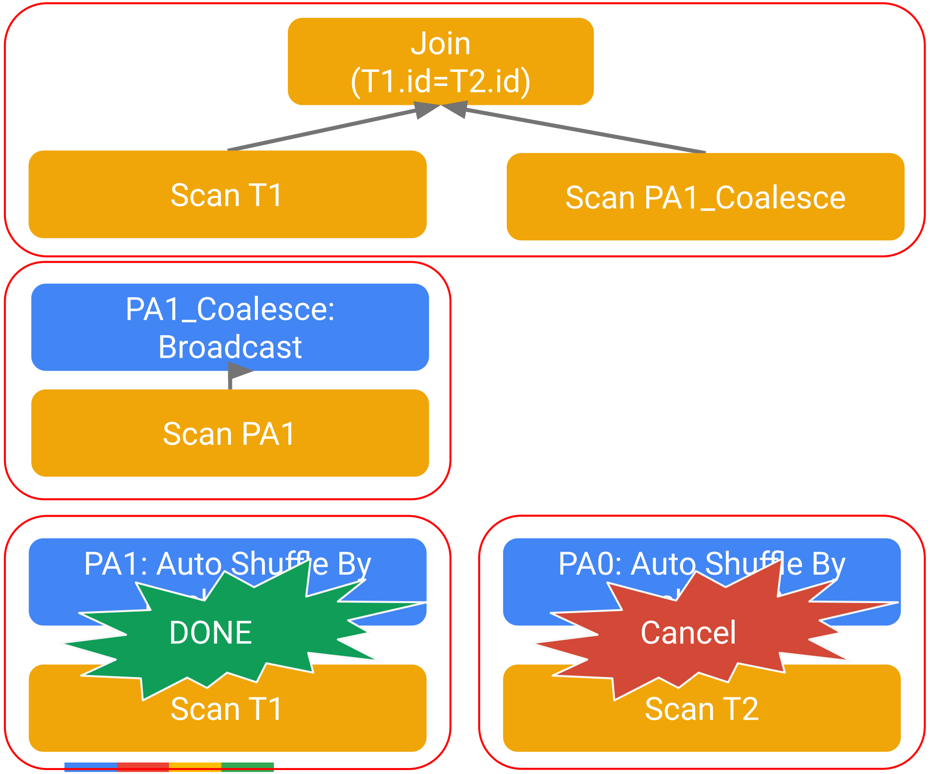


Join dynamic execution

- Decide between broadcast join and shuffled join by shuffling inputs and monitoring shuffle sizes
- Decide number of partitions for parallel join based on memory target
- Coordinate multiple joins
- Swap join in certain cases
- Coalesce stage added for broadcast join to reshuffle the initial sinks into 1
- Star and snowflake-join optimizations
    - Detect snowflake joins
    - Compute and propagate constraints predicates from dimensions to fact table
- Joins in the presence of heavy skew (heavy-hitters)
    - Repartition heavily skewed sinks using RAND(); schedule extra stages as usual

Join plan adaptation - broadcast join

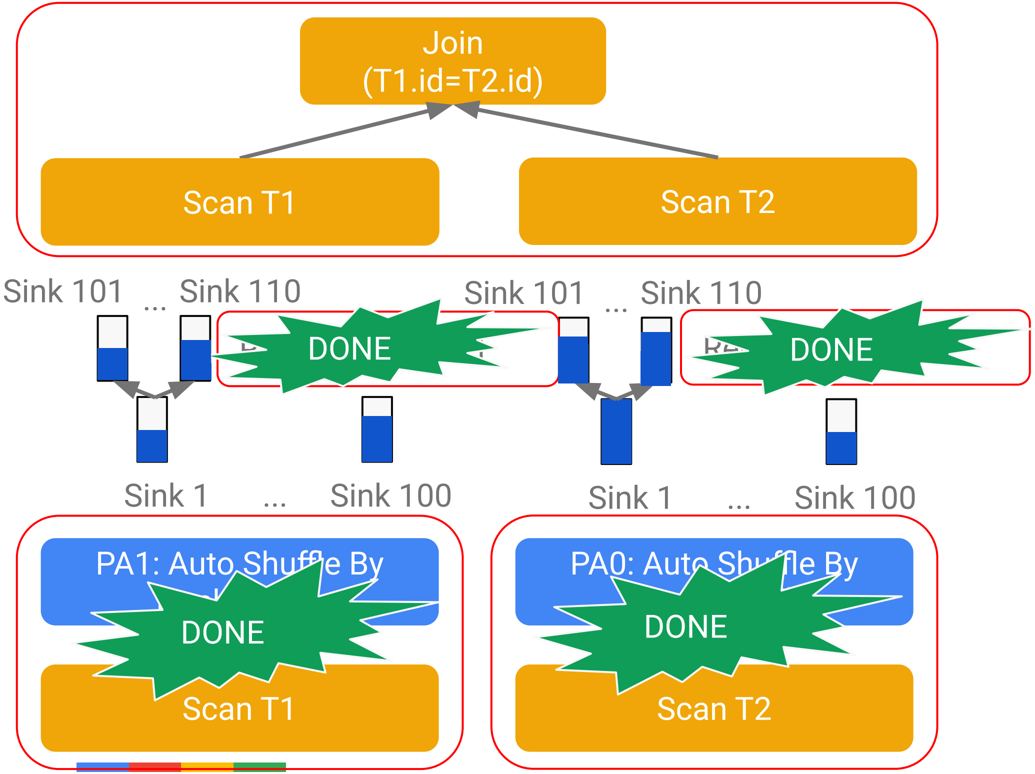
```
SELECT * FROM T1 JOIN T2 ON T1.id = T2.id
```



- Start by shuffling both inputs in 100 sinks
- Shuffling T1 finishes under Broadcast limit
- Shuffling T2 is canceled
- Coalesce the 100 sinks into 1
- Stitch plan with broadcast join
- Swap join so broadcast is on build side
- Execute join for all partitions in T2

Join plan adaptation - shuffled join

`SELECT * FROM T1 JOIN T2 ON T1.id = T2.id`



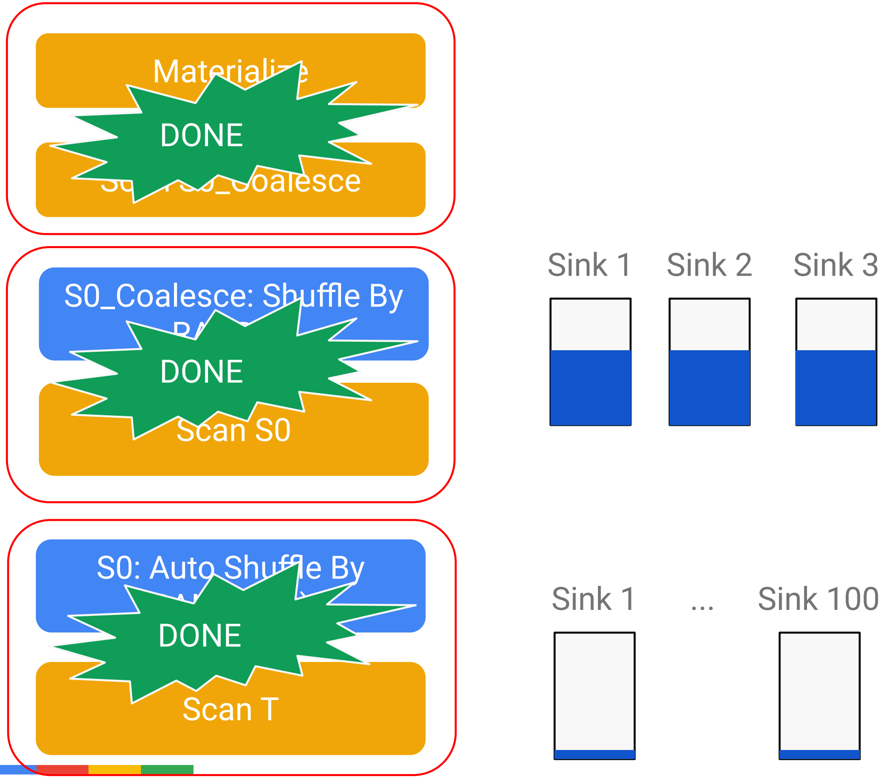
- Start by shuffling both inputs in 100 sinks
- Sink 1 from right gets over the sink limit
- Sink 1 from both sides get split into 10
- Inputs finish
- Repartition Sink 1 from both sides
- Repartitioning stages finish
- Execute join on sinks 2-110

Auto materialize/export data

- Decides the number of files for MATERIALIZE/EXPORT DATA dynamically based on the file target size
- Query planning inserts Auto Shuffle By Rand starting with f(input_partitions) sinks
- Sink limit: 1GB uncompressed data
- If sinks are too small, coalesce data using shuffle into fewer larger sinks

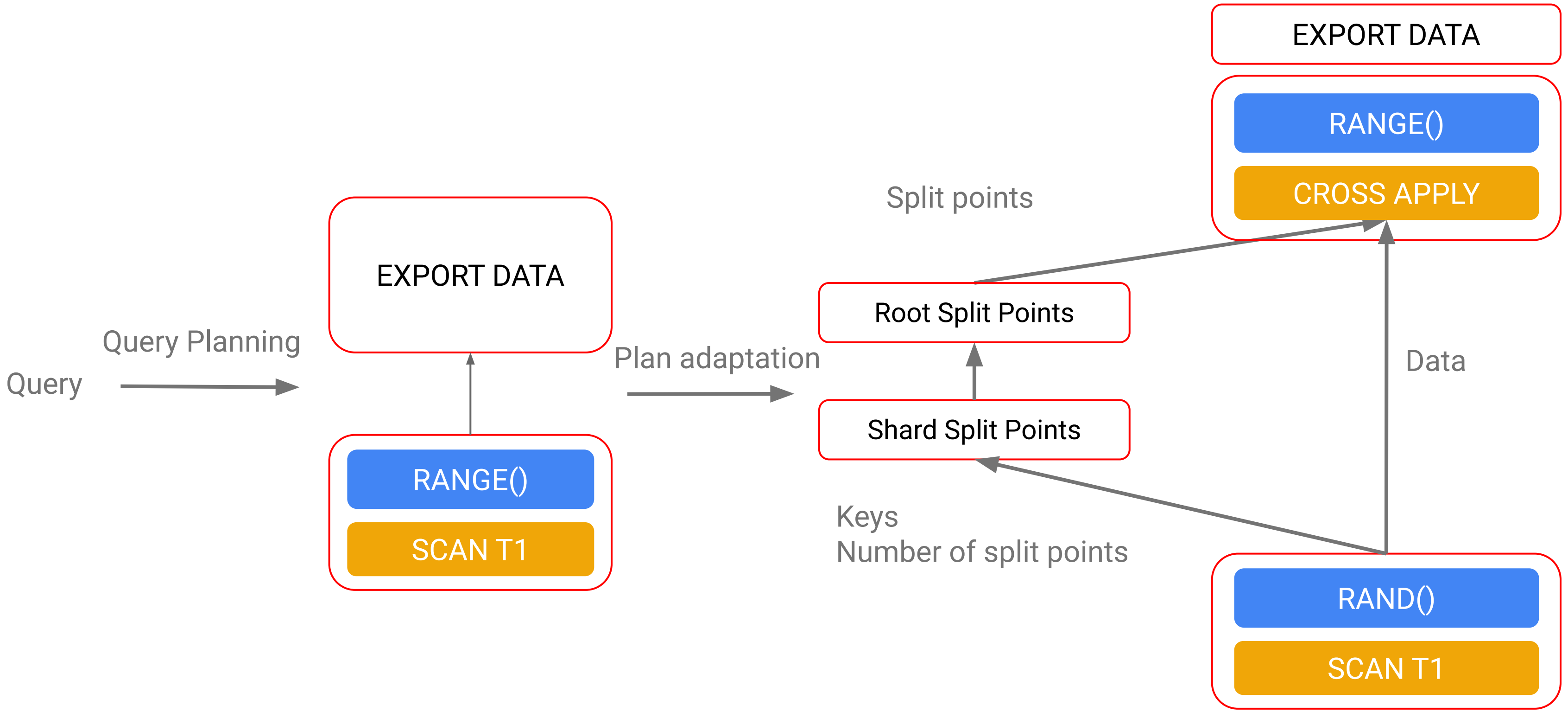
Auto materialize coalesce plan adaptation

```
EXPORT DATA ('...') AS SELECT * FROM T
```



- Start by shuffling input data in 100 sinks
- Shuffling data finishes, sinks are too small
- Coalesce data into fewer sinks
- Materialize data from the sinks
- Coalesce finishes
- Materialize finishes

Dynamic range partitioning for table clustering



Optimizing performance

- Better performance is delivered through a higher degree of parallelism
  - Clustering and partitioning increase the fan-out of scanning operations
  - Dynamic execution converges to optimal degree of parallelism
- Snowflake-join detection reduces the need for extensive join reordering
  - That being said, the query-specified join order is significant
- A lot of performance issues are solved by clustering and join order improvements
- Improvements in the next few months
  - History-based optimizations for scale-out: use past query invocations to track optimal degree of parallelism and start from that instead of converging to it through dynamic execution
  - Join-reordering: use statistics of past invocations to track join selectivities and use them to reorder joins

Questions?