Advanced Topics in Data Management

Wrap-up

Announcement

Next week, June 2nd: Project presentations

- Every team presents their project
- 10 minutes / team
- I will post the order soon
- I will post some guidelines
- Use your laptop OR my google slides
- Please come to the lecture room!

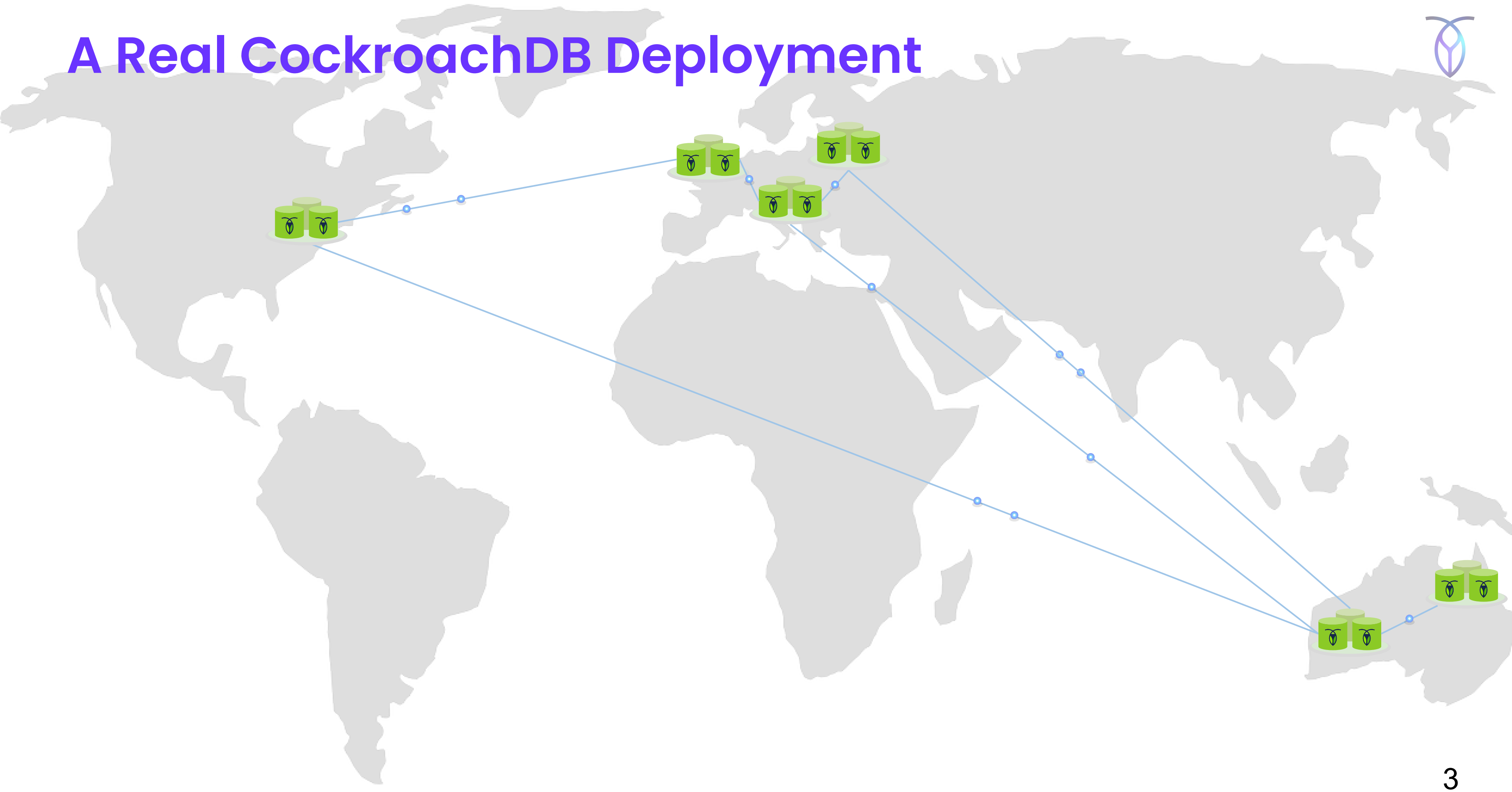
Summary

- Cockroach Lab
- Cascades
- Redshift
- Bigquery
- Teradata
- Snowflake
- RelationalAI

Cockroach Lab

Cockroach Lab

A Real CockroachDB Deployment



Cockroach Lab

CockroachDB's First Optimizer

- Not an optimizer
- Used heuristics (rules) to choose execution plan
- E.g. "if an index is available, always use it"
- E.g. "always use the index, except when the table is very small or we expect to scan more than 75% of the rows, or the index is located on a remote machine"
- Sort of works for OLTP, but customers run everything

Cockroach Lab

Phases of plan generation

Cockroach Lab

Phases of plan generation

Cockroach Lab

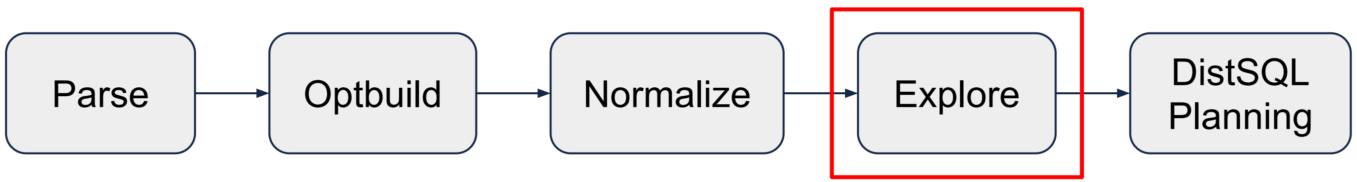
Normalization rules

- Transformation rules create a logically equivalent relational expression
- Normalization (or “rewrite”) rules are “always a good idea” to apply
- Examples
  - Eliminate unnecessary operations: `NOT (NOT x) -> x`
  - Canonicalize expressions: `5 = x -> x = 5`
  - Constant folding: `length(‘abc’) -> 3`
  - Predicate push-down*
  - De-correlation of subqueries*
  - ...

* Not always a good idea, but almost always

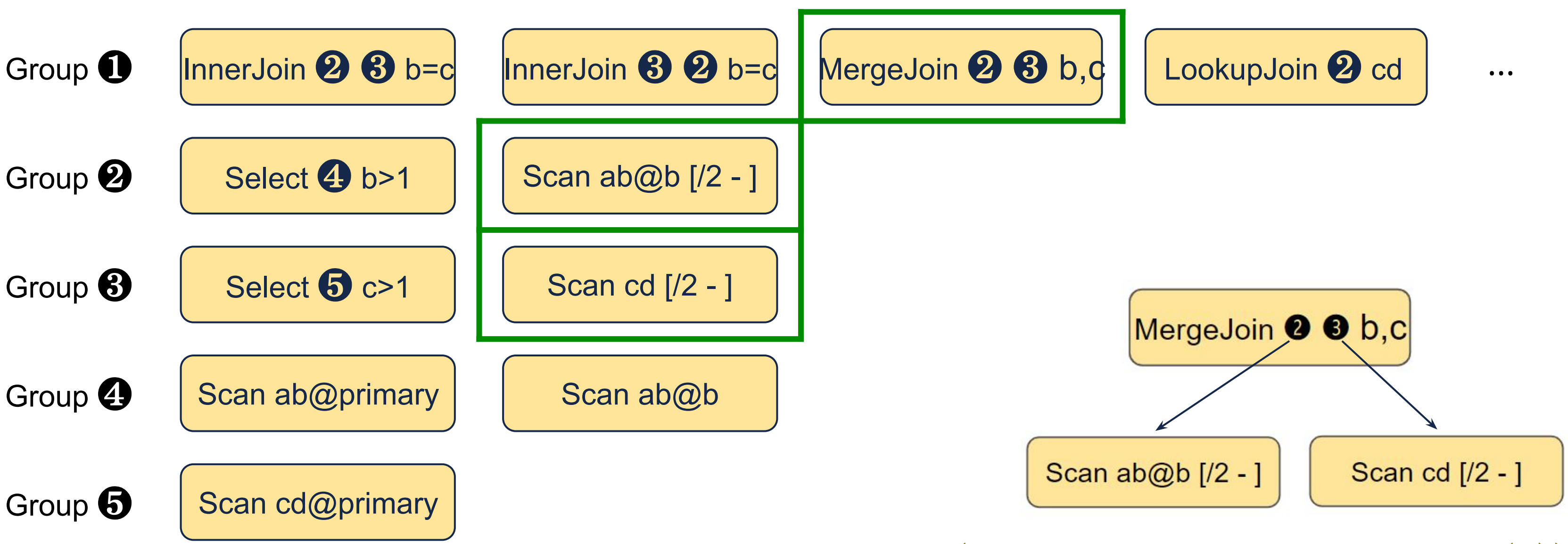
Cockroach Lab

Phases of plan generation



Cockroach Lab

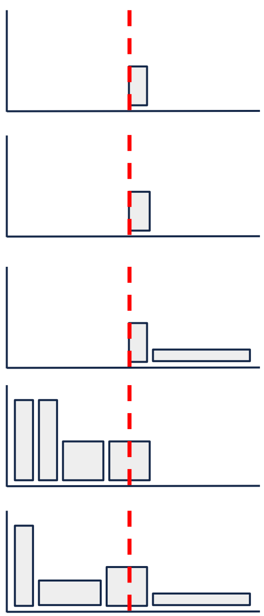
Explore: GenerateLookupJoins




```sql
CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
```

Cockroach Lab

Calculate Statistics



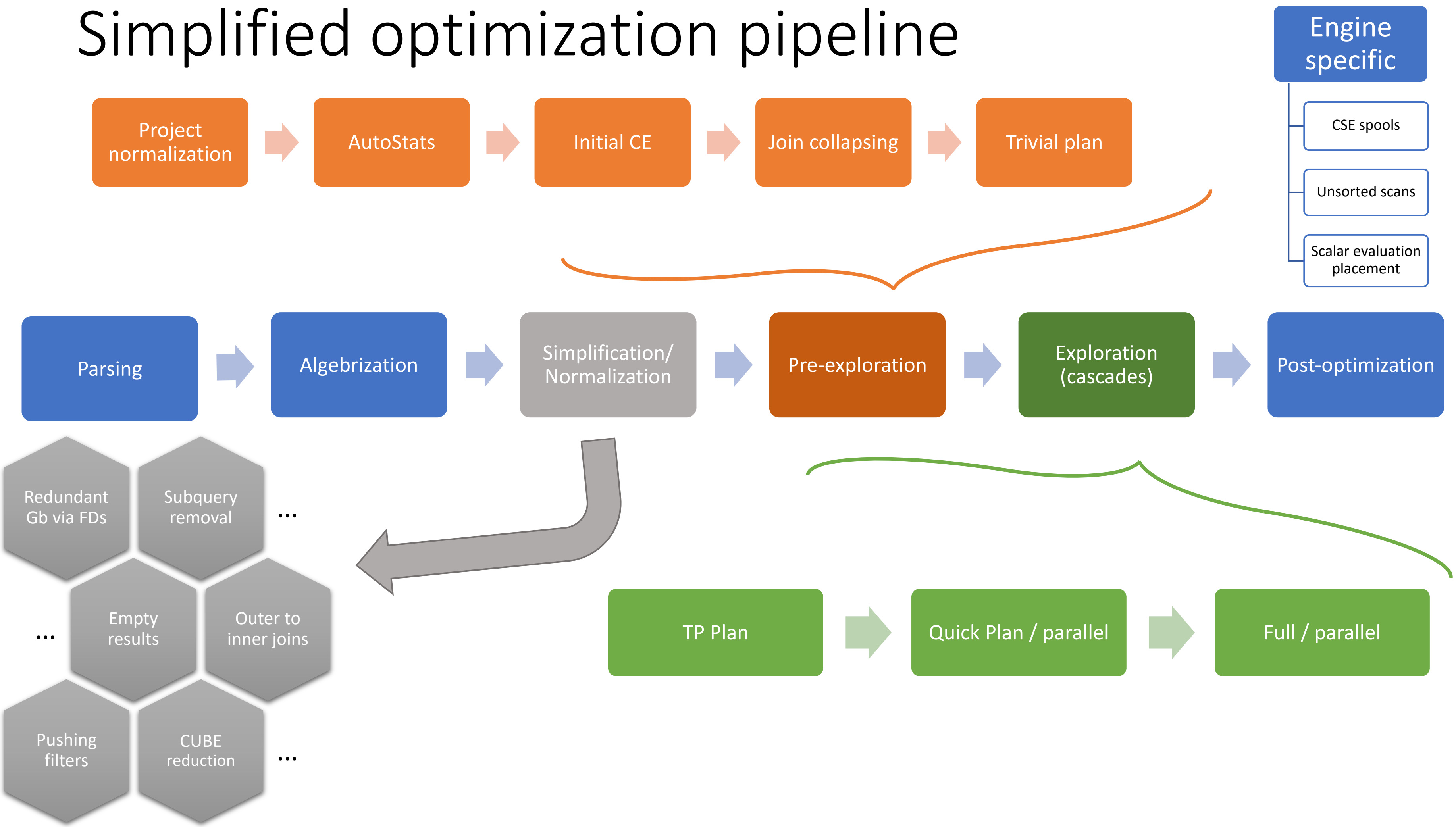
Hist(b,c)	Rows: 500	Group ❶	InnerJoin ❷ ❸ b=c	InnerJoin ❸ ❷ b=c	MergeJoin ❷ ❸ b,c	...
Hist(b)	Rows: 500	Group ❷	Select ❹ b>1	Scan ab@b [/2 -]		
Hist(c)	Rows: 1500	Group ❸	Select ❺ c>1	Scan cd [/2 -]		
Hist(b)	Rows: 4000	Group ❹	Scan ab@primary	Scan ab@b		
Hist(c)	Rows: 4000	Group ❺	Scan cd@primary			

```sql
CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
```

Cascades

Cascades

Simplified optimization pipeline



Cascades

Rules & Properties

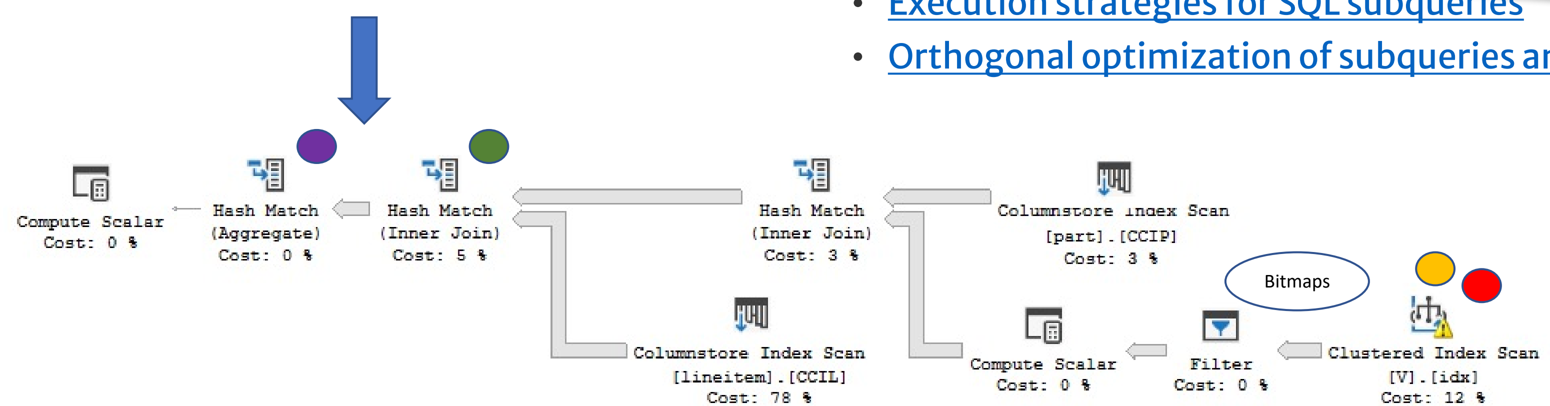
```sql
select sum(l_extendedprice) / 7.0 as avg_yearly
from lineitem join part on p_partkey = l_partkey
where
    p_brand = 'Brand#12'
    and l_quantity < (select 0.2 * avg(l_quantity)
                      from lineitem
                      where l_partkey = p_partkey)
```

```sql
create view V with schemabinding as
select l_partkey, sum(l_quantity) sc, count_big(*) cb
from dbo.lineitem
group by l_partkey
```

400+ rules

- Join reordering
- Outer joins
- Subqueries
- Aggregation
- Union
- Stars and snowflakes
- Join elimination
- Empty table simplification
- Materialized views
- Index plans
- Large IN lists
- Update plans
- Halloween protection
- Partitioned tables
- Parallelism
- Remote queries
- ...

- Execution strategies for SQL subqueries
- Orthogonal optimization of subqueries and aggregation



Cascades

Statistics

Taxonomy

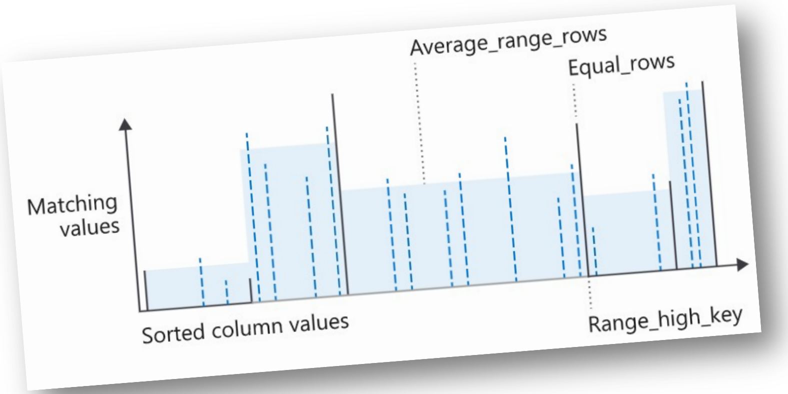
- Single-column 'MaxDiff' histograms
- Multi-column density information
- Average column lengths
- Tries
- HLL / Heavy Hitter sketches (DW / Partitioned tables)
- Skew (Cosmos)

Data sources

- Base tables (including computed columns)
- Filtered indexes
- Materialized views

Create / Update mechanics

- Creation: manual, implicit, automatic
- Update: manual, automatic with mod counts
- Block-level sampling (optional cross-validation)

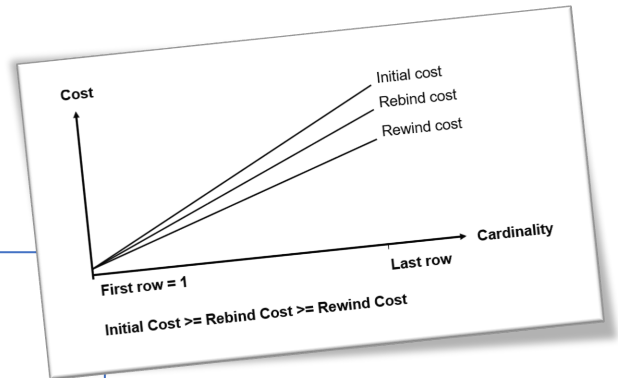


Cascades

Costing

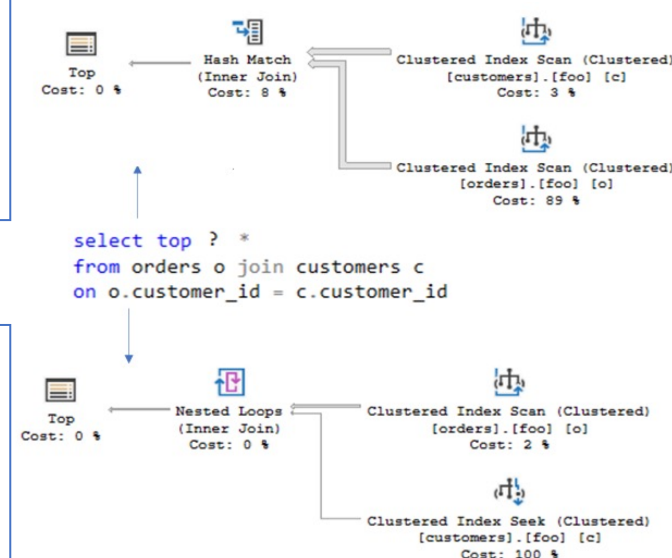
Bottom-up calculation…

- CPU (e.g., filters) and I/O (e.g., spilling aggs)
- Information: CE, DV, outliers, row sizes, DOP, memory, sorted-ness, etc.
- 3 cost lines: Initial / rewind / rebind



… with top-down context

- Row goals
- Bitmap filters
- Estimated rewinds/rebinds



Decouple Logical / Physical

Logical optimization = equality saturation (Egg)

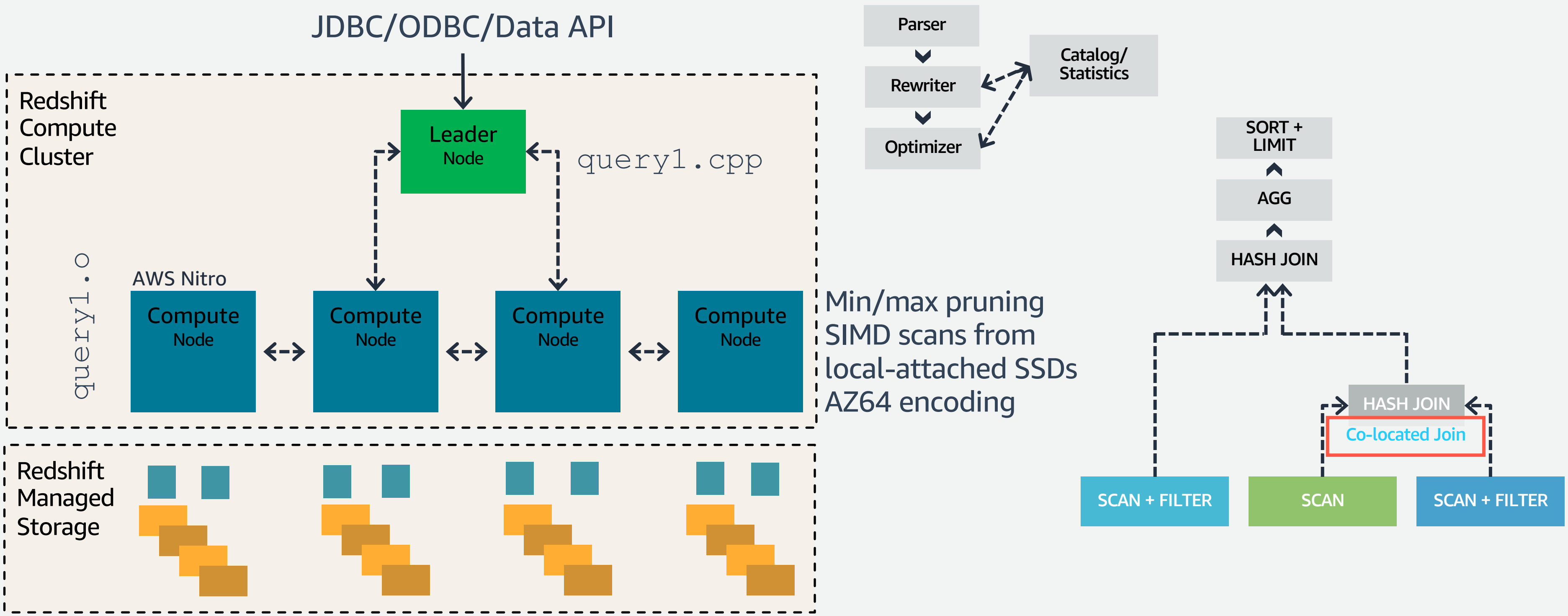
Physical optimization:

- Optimize(A join B)
  - A MergeJoin B:
    - Optimize(A, sort, cost < infty)
    - Optimize(B, sort, cost < infty)
    - Total cost = **100**
  - A HashJoin B
    - Optimize(A, -, **cost < 100**)
    - Optimize(B, -, **cost < 100**)

Redshift

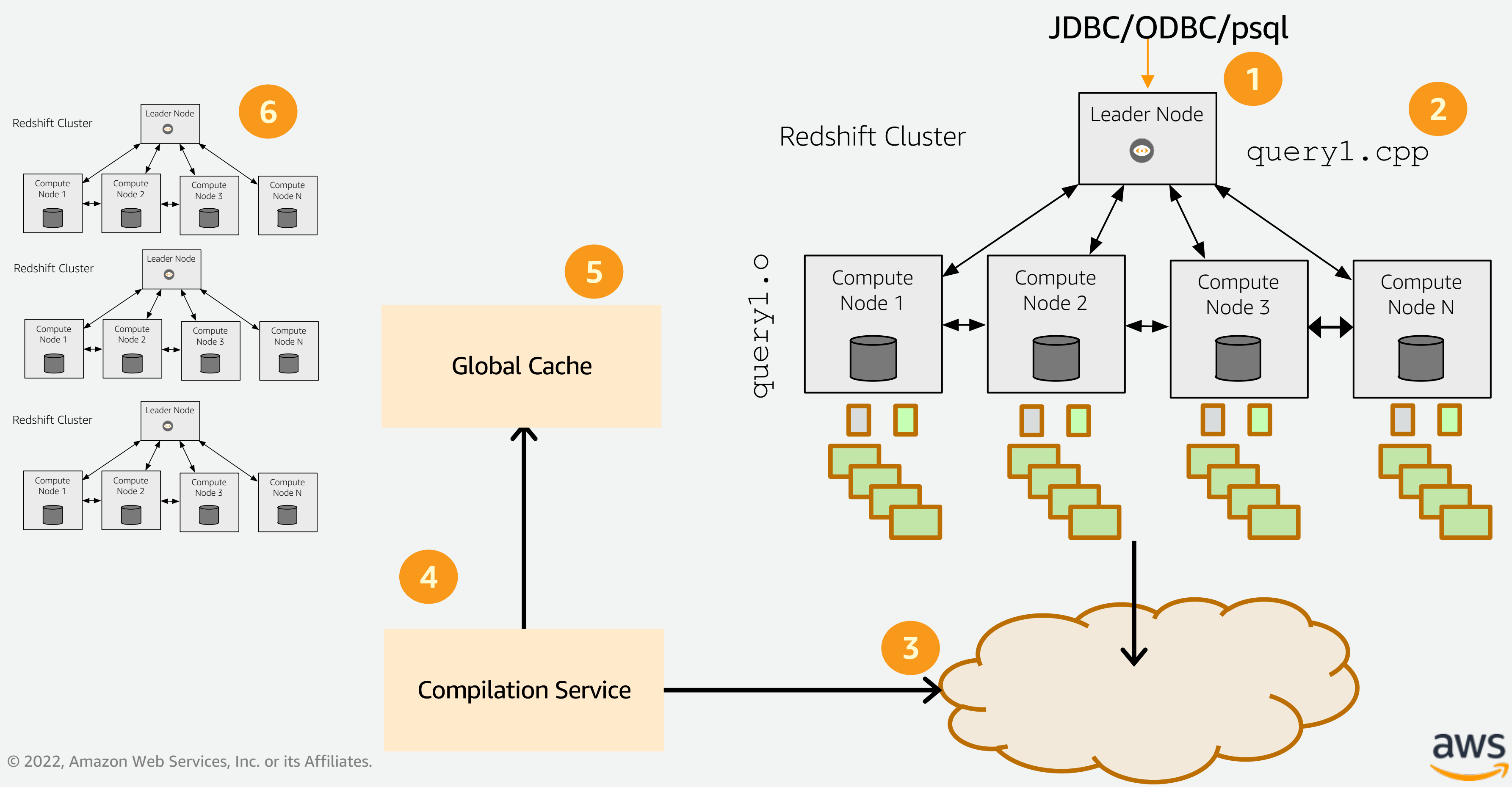
Redshift

Executing a query in Amazon Redshift



Redshift

Compilation-as-a-Service



Detour: Push v.s. Pull

$$\Gamma_{A,sum(B)}$$

$$\sigma_{P2}$$

$$\sigma_{P1}$$

$$R$$

Push

```
for x in R do:
  if P1(x) then
     if P2(x) then
        insert(x,hashtable)
```

Pull

```
repeat       // Gamma asks for next()
  repeat     // sigma_p2 asks for next()
    repeat   // sigma_p1 asks for next()
      x = R.next()
    until x == NULL or P1(x)
  until x == NULL or P2(x)
  if x != NULL: insert(x,hashtable)
until x == NULL
```

Redshift

Ingesting and Querying Semistructured Data

with the SUPER encoding & the PartiQL Query Language

- Rapid insertion of flexible, schemaless JSON data
- Efficient, navigation-friendly Redshift SUPER encoding
- Flexible PartiQL queries for discovery
- PartiQL extends SQL with “first class citizen” nested data and dynamic typing
- PartiQL materialized views extract, load & transform (ELT) from SUPER

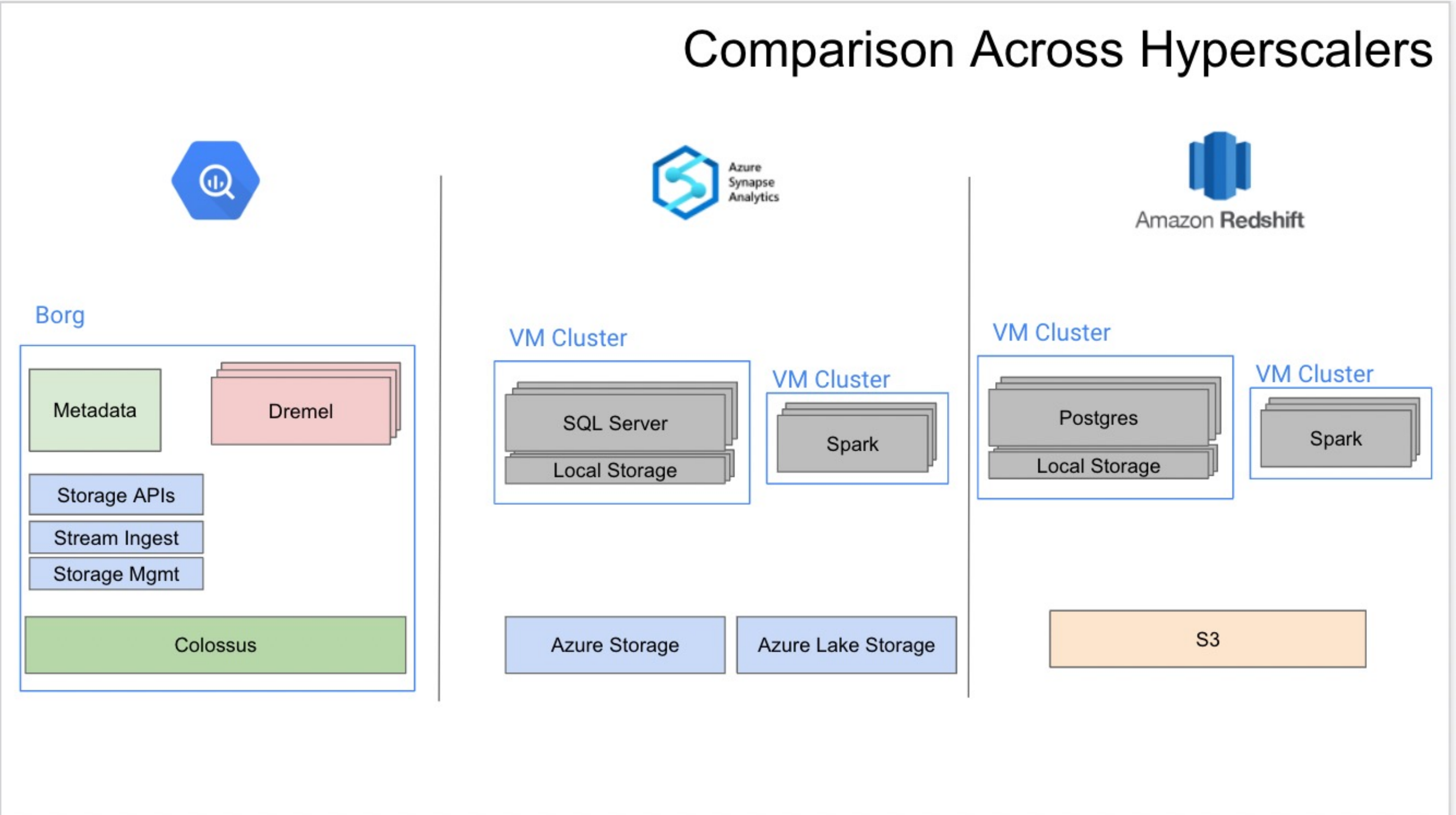

```
{
"id":1,
"name":{"given":"Jane", "family":"Doe"},
"phone":[{"type":"work", "num": "9252364000"},
         {"type":"cell", "num": 6501234444}]
}
{
"id":2,
"name":{"given":"Graham", "family":"Bell"},
"phone":[{"type":"work", "num": 5106101234}]
}
```


```
SELECT name.given AS firstname, ph.num
FROM customers c, c.phone ph
WHERE ph.type = 'cell';

firstname | num
----------+---------------
"Jane"    | 6501234444
```

BigQuery

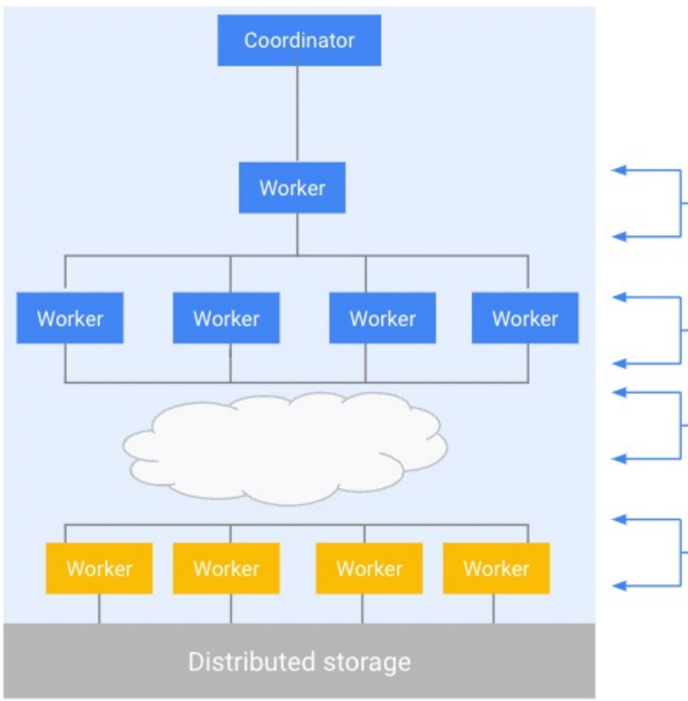
BigQuery


Comparison Across Hyperscalers
Azure Synapse Analytics
Amazon Redshift
Borg
Metadata
Dremel
Storage APIs
Stream Ingest
Storage Mgmt
Colossus
VM Cluster
SQL Server
Local Storage
VM Cluster
Spark
Azure Storage
Azure Lake Storage
VM Cluster
Postgres
Local Storage
VM Cluster
Spark
S3

BigQuery

Flexible Query Execution

```
SELECT language, MAX(views) as views
FROM `wikipedia_benchmark.Wiki1B`
WHERE title LIKE "G%o%"
GROUP BY 1 ORDER BY 2 DESC LIMIT 100
```



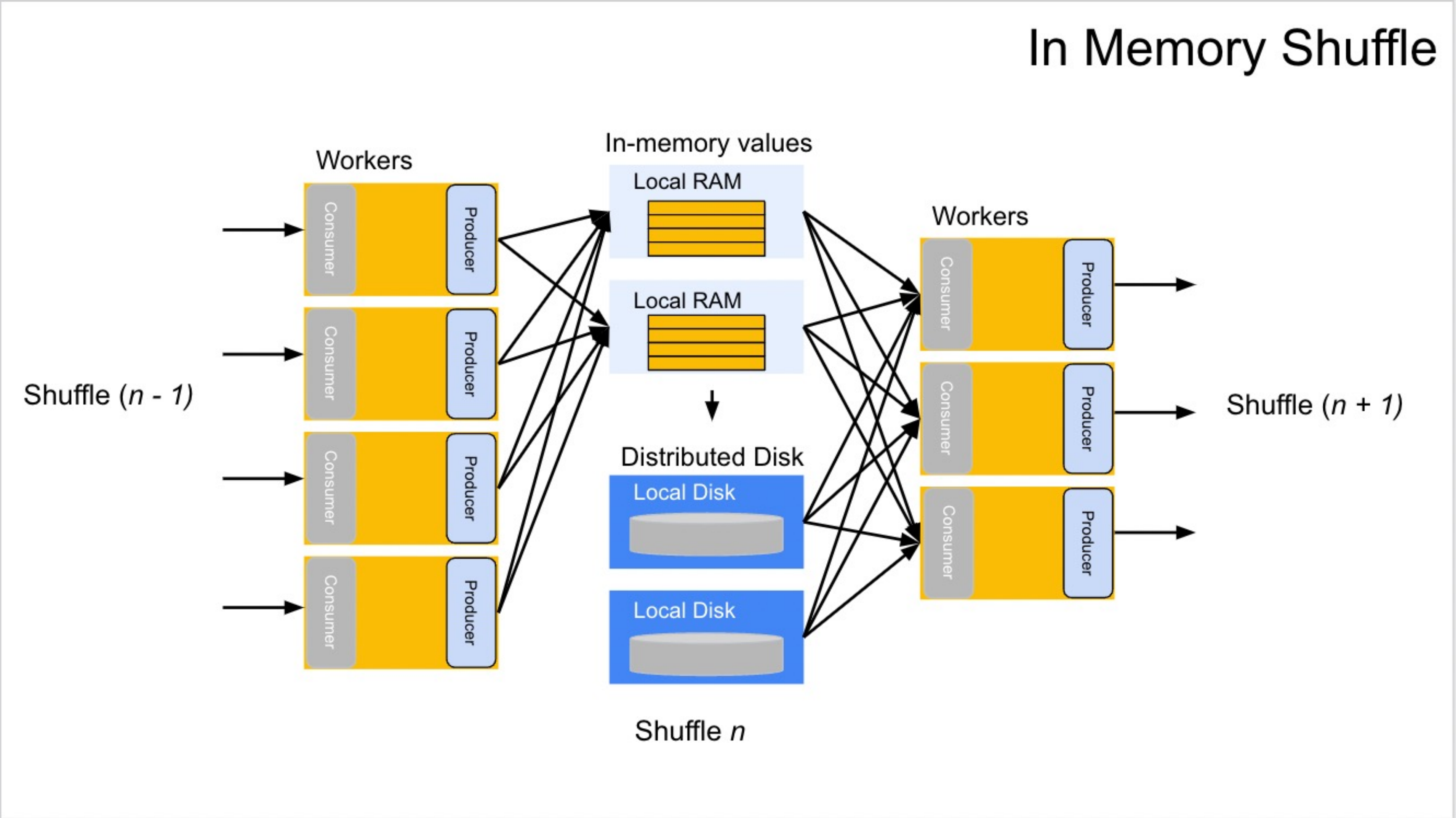
Stage 3: SORT, LIMIT

Stage 2: GROUP BY, SORT, LIMIT

Shuffle with dynamic partitioning

Stage 1: Partial GROUP BY

BigQuery


In Memory Shuffle
Workers
In-memory values
Local RAM
Local RAM
Workers
Consumer
Producer
Consumer
Producer
Consumer
Producer
Consumer
Producer
Shuffle (n - 1)
Distributed Disk
Local Disk
Local Disk
Consumer
Producer
Consumer
Producer
Consumer
Producer
Shuffle (n + 1)
Shuffle n

BigQuery

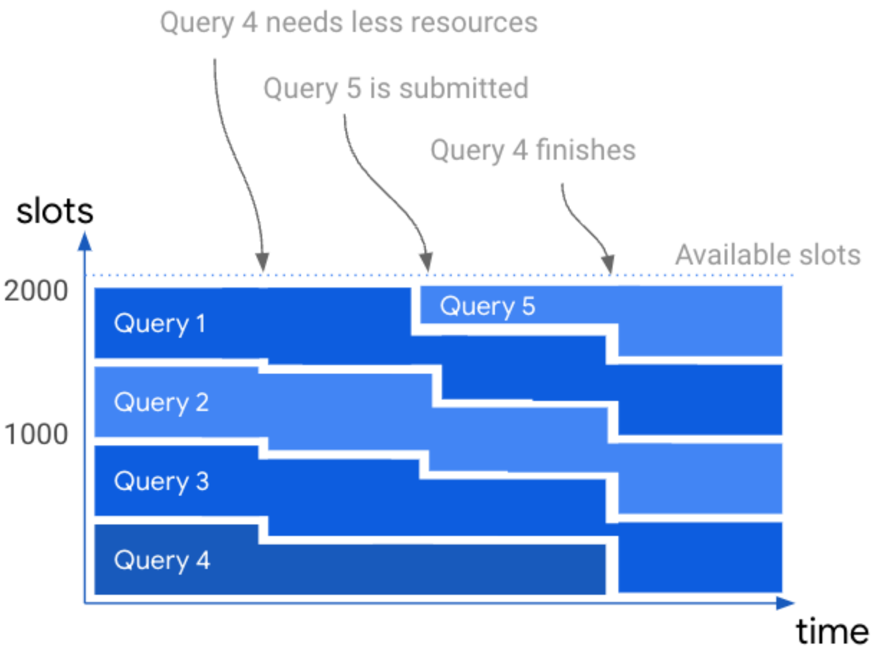
In Memory Shuffle Details

- **In-memory shuffle coupled with compute presents bottlenecks**
  - Hard to mitigate quadratic scaling characteristics
  - Resource fragmentation, stranding, poor isolation
- **BigQuery implements a disaggregated memory-based shuffle**
  - RAM/disk managed separately from compute tier
  - Reduced shuffle latency by order-of-magnitude
  - Enables order-of-magnitude larger shuffles
  - Reduced resource cost by 20%
- **Persistence in shuffle layer**
  - Checkpoint query execution state
  - Allows flexibility in scheduling + execution (preemption of workers)

BigQuery

Dynamic Scheduling in BigQuery

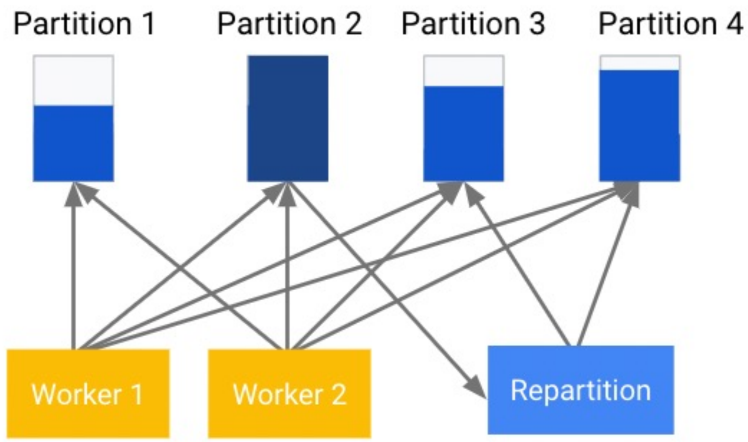
- **Dynamic central scheduler allocates**
  - Slots
  - Workers
- **Handles machine failure**
- **Fair resource distribution between queries**



BigQuery

Dynamic Partitioning

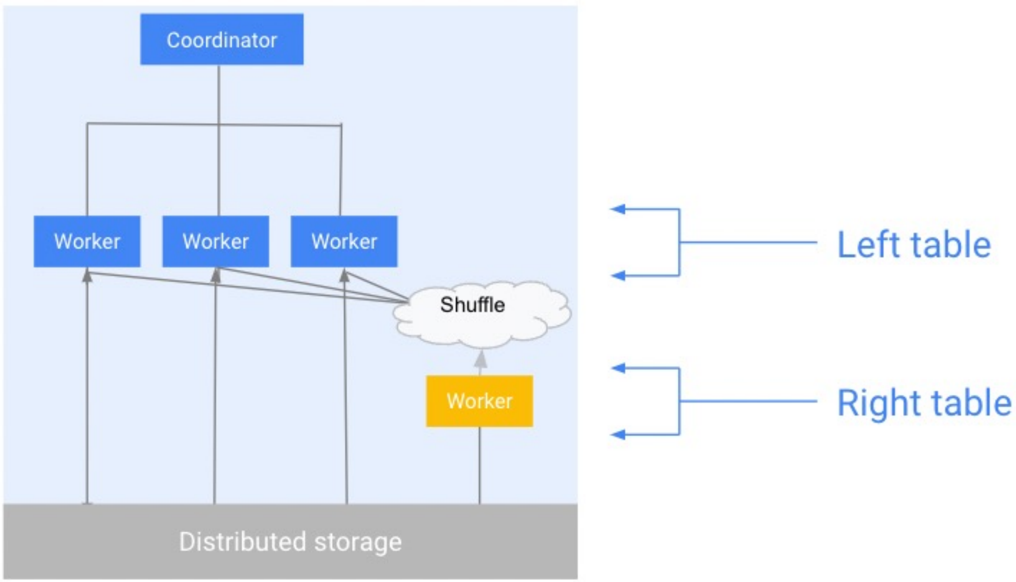
Goal: Dynamically load balance and adjust parallelism while adapting to any query or data shape and size



- Workers start writing to Partitions 1 and 2
- Query Coordinator detects there is too much data going to Partition 2
- Workers stop writing to Partition 2 and start writing to Partitions 3 and 4
- Data in Partition 2 is re-partitioned into Partitions 3 and 4

BigQuery

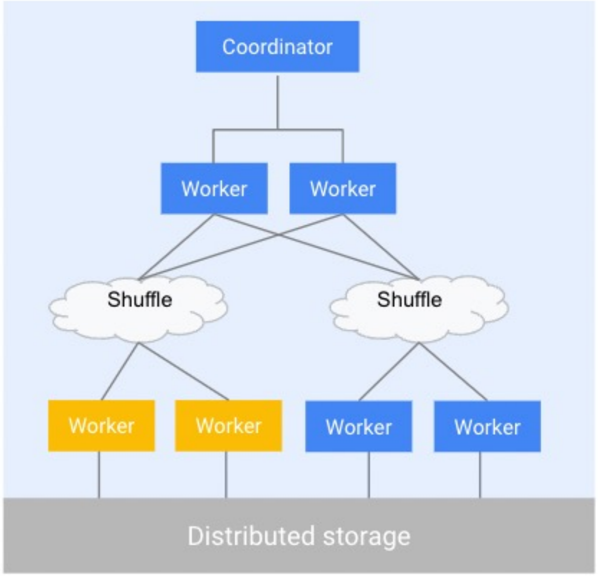
Broadcast Join




```
SELECT
  c.author.name a, c2.a m
FROM github_repos.commits c
JOIN (
  SELECT
      committer.name a,
      commit
  FROM
github_repos.commits) c2
ON
  c.commit = c2.commit
WHERE c2.a = 'tom'
LIMIT 1000
```

BigQuery

Shuffle Join



Hash join

Independent shuffles

```
SELECT c.author.name a,  c2.a m
FROM github_repos.commits c
JOIN (SELECT committer.name a, commit
      FROM github_repos.commits) c2
ON c.commit = c2.commit
LIMIT 1000
```

BigQuery

Dynamic Join Processing Examples

- Start with hash join by shuffling data on both sides
  - Cancel shuffle one side finished fast and is below a broadcast size threshold
  - Execute broadcast join instead (much less data transferred)
- Decide number of partitions/workers for parallel join based on input data sizes
- Swap join sides in certain cases
- Star schema join optimizations
  - Detect star schema joins
  - Compute and propagate constraint predicates from dimensions to fact table

Teradata

Teradata

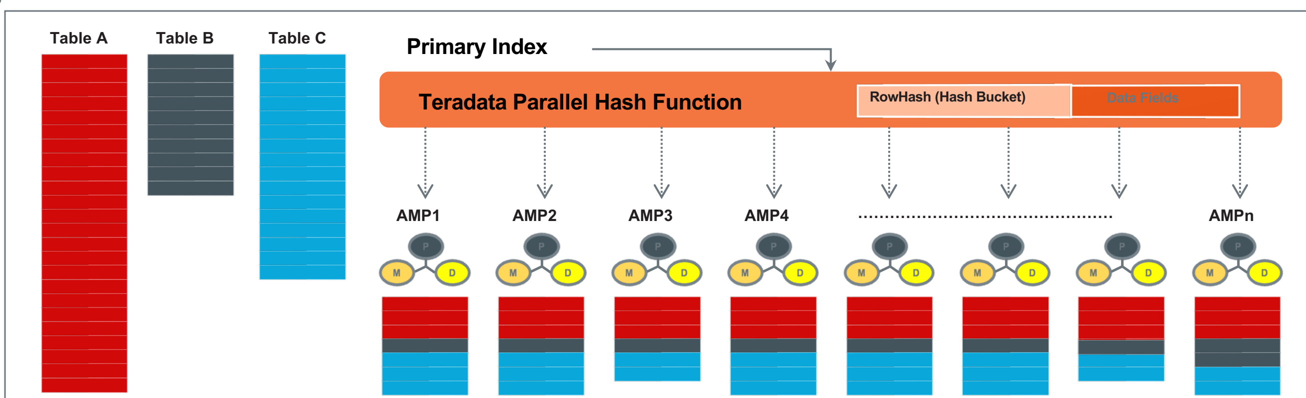
Teradata Data Management

Data Management

Rows automatically distributed evenly by **hash partitioning**

- Even distribution results in scalable performance
- Done in real-time as data are loaded, appended, or changed.
- Hash map defined and maintained by the system
  - 2**32 hash codes, 1,048,576 buckets distributed to AMPs

- Primary Index (PI) column(s) are hashed
- Hash is always the same - for the same values
- No reorgs, repartitioning, space management



Teradata

Defining a Table in Teradata

Create the table

- Standard SQL syntax

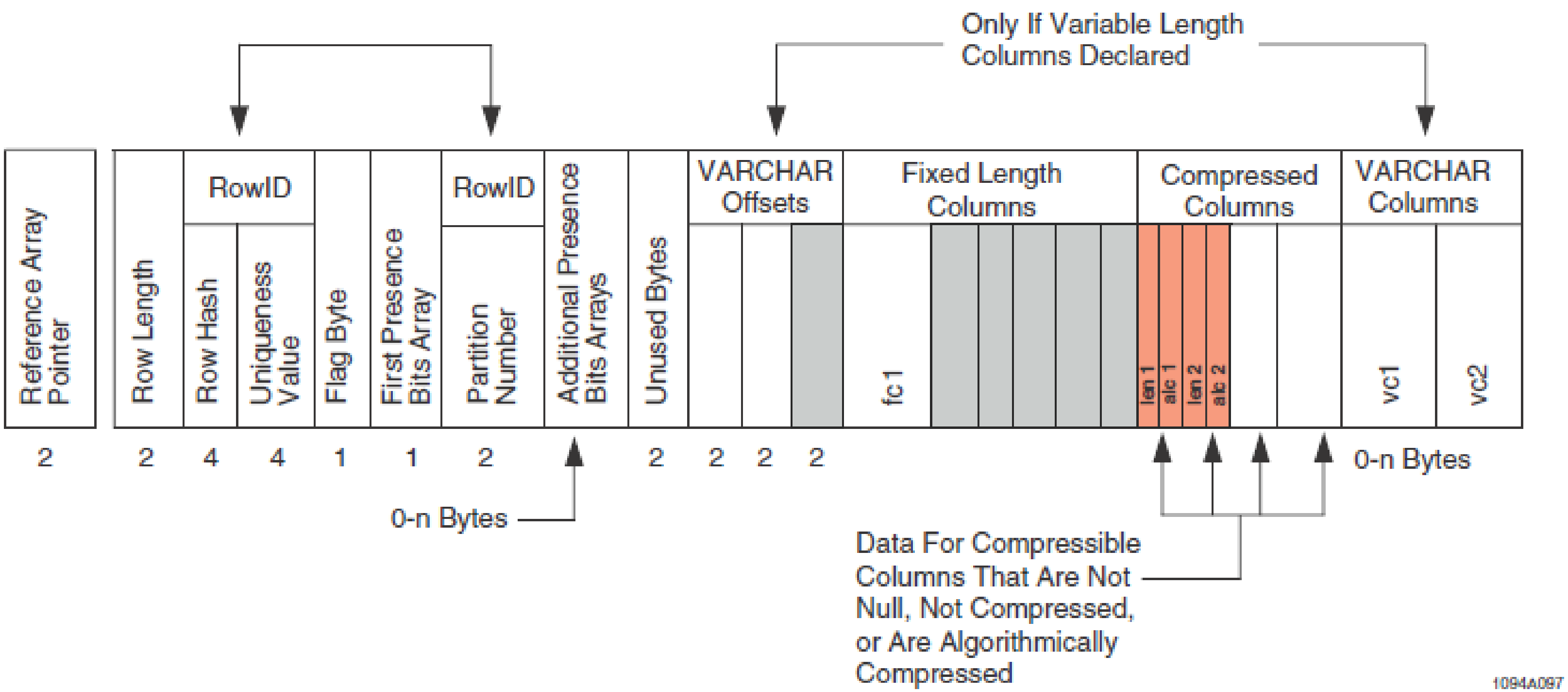
Define the primary index

- Extra line at end of table definition

```
CREATE TABLE LineItem (
        OrderKey INTEGER NOT NULL,
        PartKey INTEGER NOT NULL,
        SupplierKey INTEGER,
        LineNumber INTEGER,
        Quantity INTEGER NOT NULL,
        ExtendedPrice DECIMAL(13,2),
        Discount DECIMAL(13,2),
        Tax DECIMAL(13,2),
        Comment VARCHAR(50)
        )
PRIMARY INDEX (OrderKey);
```

Teradata

Base Table Row Formats

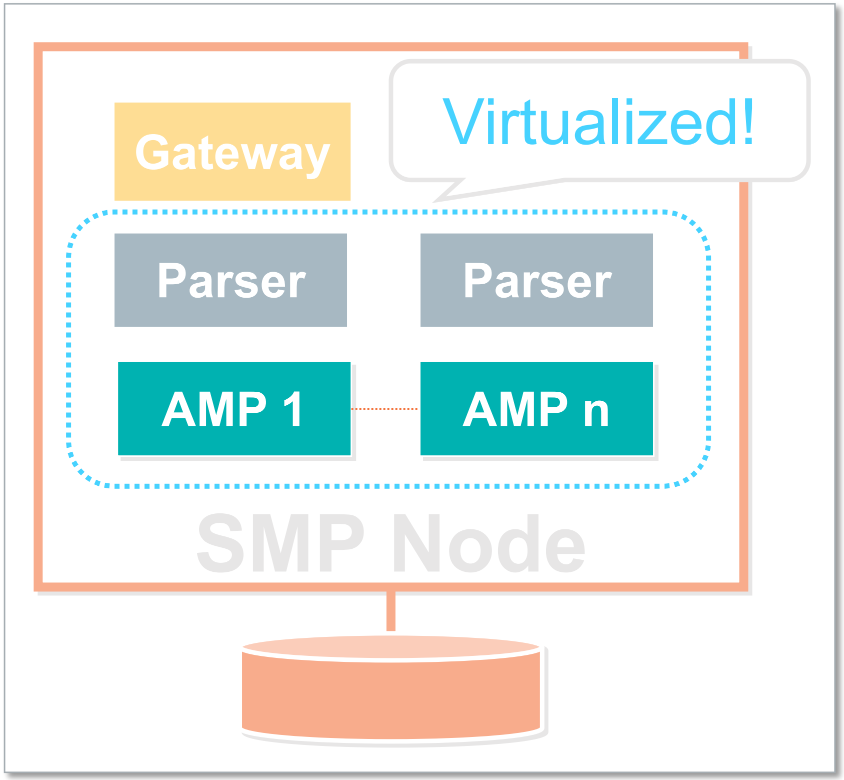


Teradata

What's on a Node

- **Gateway**
  - Connect sessions to outside world
  - Balance external traffic workload across nodes
- **Parsing Engine (PE)**
  - Parse & Optimize
  - Dispatcher to AMPs
- **AMP (Access Module Processor)**
  - Execution engine
  - Logs & locks
  - Data dictionary
  - I/O management
- **"Vprocs"**
  - Virtual "processors" sharing one physical node

Query Execution



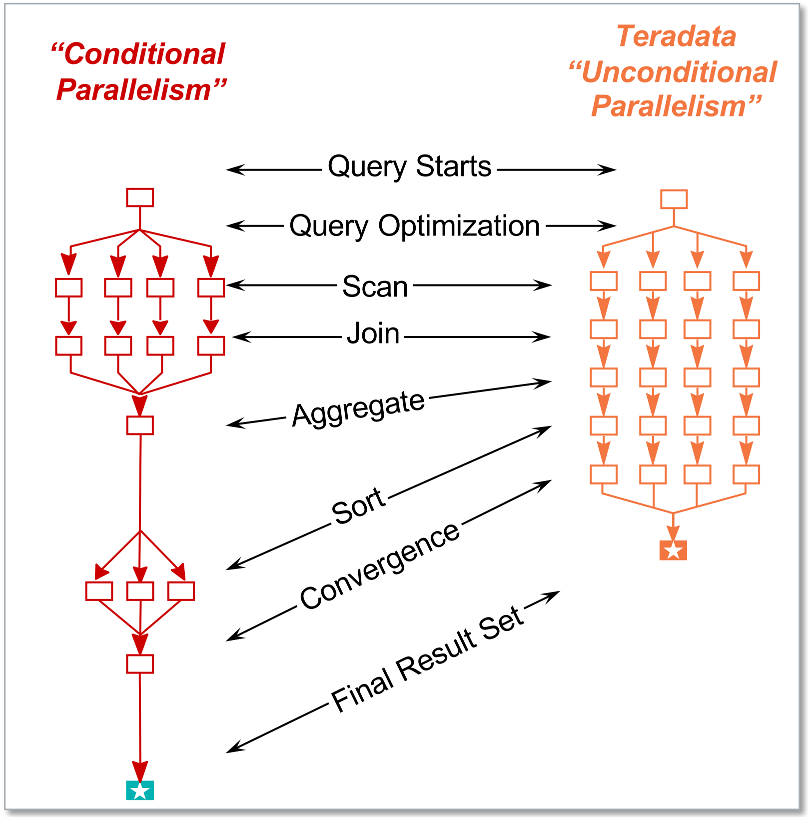
teradata.

Teradata

Query Parallelization

Query Execution

- Query parsing, management is fully distributed across the nodes
  - No head node/coordinator node
- All operations fully parallel
  - No single threaded operations
  - Scans, Joins, Index access, Aggregation, Sort, Insert, Update, Delete
  - Ordered Analytics
  - Extensibility functions
  - Result return



teradata.

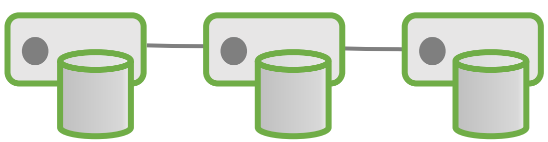
Snowflake

Snowflake

TRADITIONAL DATABASE ARCHITECTURES

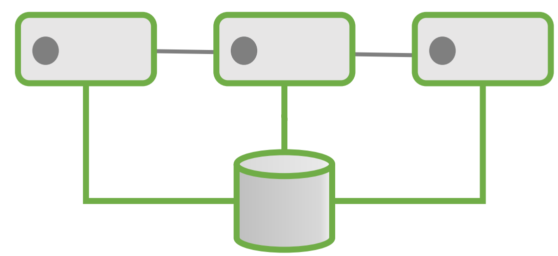
Limited Scalability, Not Elastic

Shared-nothing



- Distributed Storage
- Single Cluster
- Adopted by Gamma, Teradata, Redshift, Vertica, Netezza, …

Shared-disk

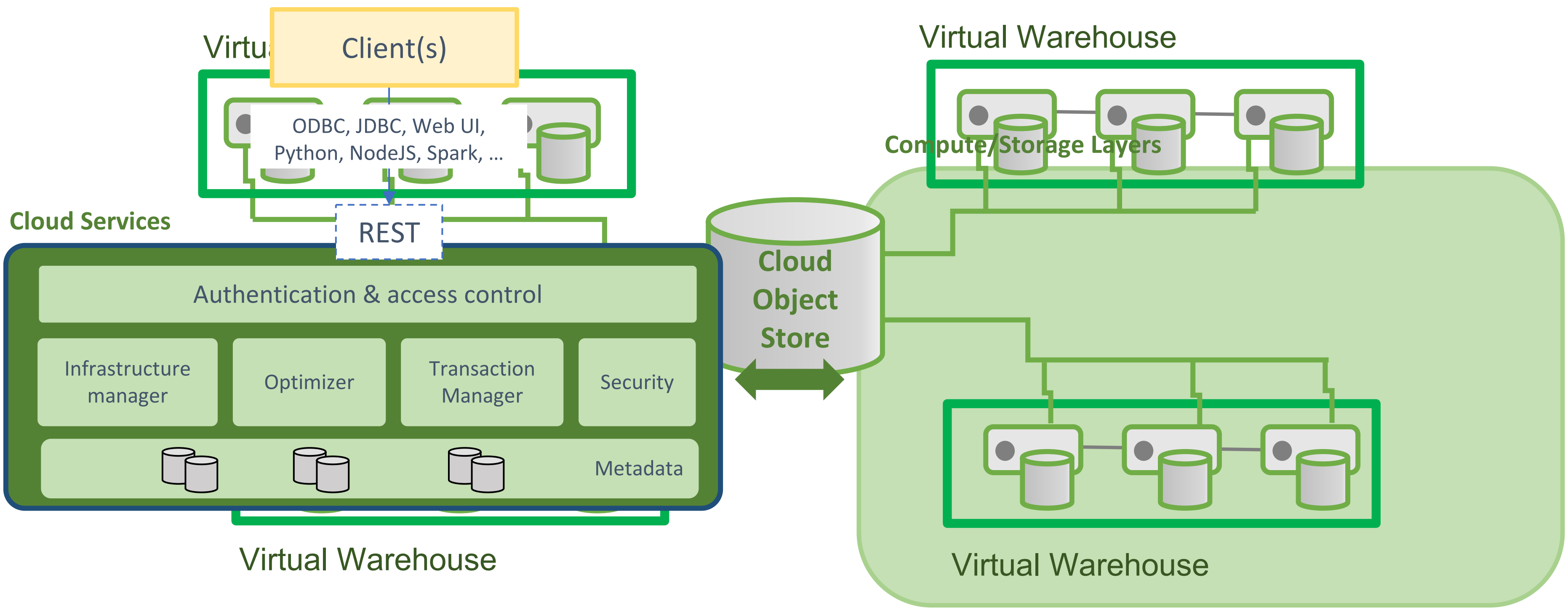


- Centralized Storage
- Single Cluster
- Adopted by Oracle, Hadoop

Snowflake

SNOWFLAKE REGION ARCHITECTURE

Multi-cluster, Shared-data



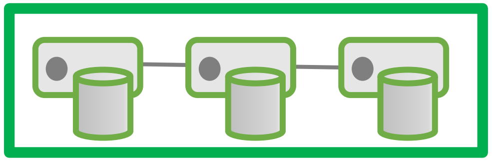
Snowflake

STORAGE TIER

- **Immutable Storage**
  - Each table is automatically partitioned horizontally
  - Partition size is kept very small, generally 16MB
  - Each partition is backed by an immutable file
  - Partitions are columnar organized, compressed, encrypted
  - Partitions are the unit of change for transactions
- **Semi-structured**
  - Variant data type used to store schemaless semi-structured data
  - Automatic columnarization of semi-structured attributes
- **Partition Metadata**
  - Out-of-box, metadata is automatically stored for all columns/sub-columns in a partition
  - Leverage that metadata to perform partition pruning
  - Re-clustering service to improve pruning
  - Track all table mutations to provide full ACID support

Snowflake

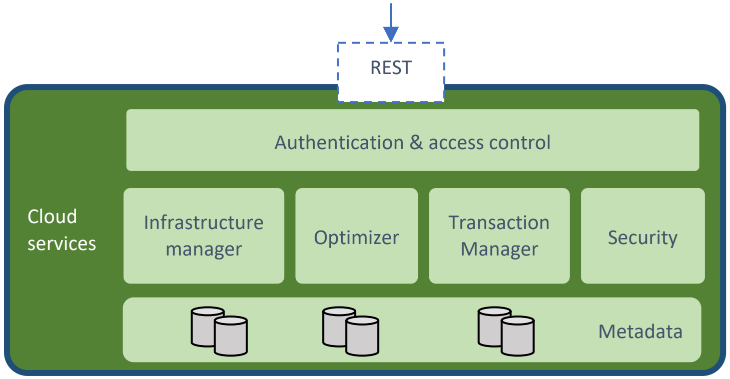
COMPUTE TIER



- **Virtual warehouse**
  - Snowflake Entity used to manage the set of compute resources used by a workload
  - Made of one or more compute clusters
  - Cluster size range from one to several hundred nodes
  - Workloads are fully isolated from each other
- **Just-in-time Compute**
  - Sub-second auto-resume when associated workload starts
  - Online resize up and down possible while workload runs
  - Auto-suspend when workload is no longer running
  - Snowflake charges usage by second of compute resource used
    ➔ **FAST is free!**
- **Partition Cache**
  - Node local memory and SSD storage used to cache partitions
  - Only columns/sub-columns which are accessed are cached
  - Highly available, fully stateless

Snowflake

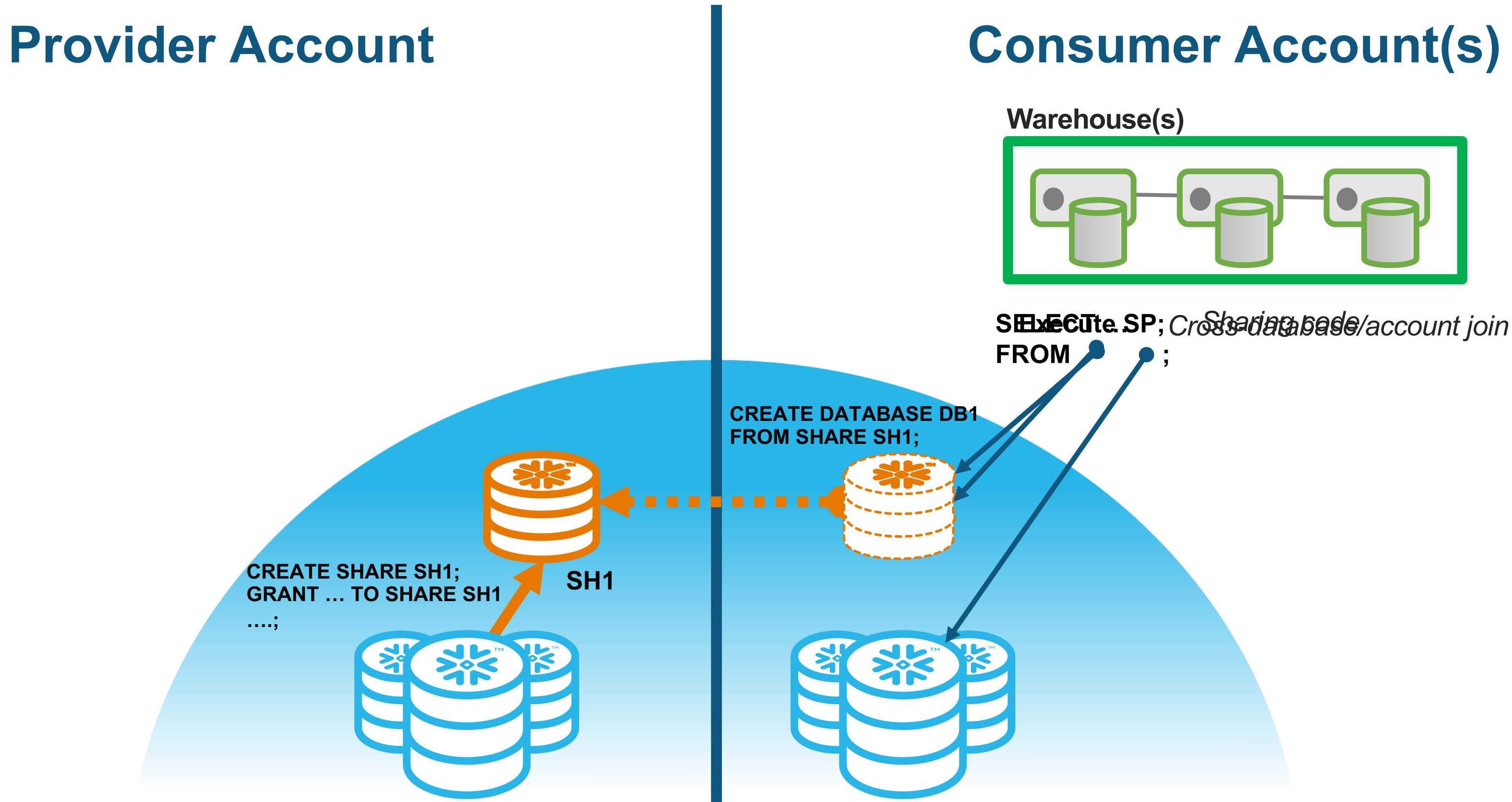
CLOUD SERVICES



- **Control Plane of a Snowflake Region**
  - Connection Management
  - Infrastructure Provisioning and Management
  - Metadata storage (use FDB) & management
  - Query planning and optimization
  - Transaction management
  - Security management
- **Self-managed**
  - Self-upgrade of both software and hardware
  - Self-healing: replacement of failed servers and transparent re-execution of any failed queries
  - Highly available over multiple availability zone
  - Stateless: persistent sessions for load-balancing and transparent fail-over

Snowflake

SNOWFLAKE DATABASE SHARING



Final Thoughts

- Common themes:
  - Optimization
  - Execution
  - parallelism
- New directions:
  - Tensors
  - ML
  - Global Distribution