Query optimizer implement

ORCA&MEMSQL

quxing

tianjiqx@126.com

Part I:

ORCA:A Modular Query Optimizer Architecture for Big Data

Motivation

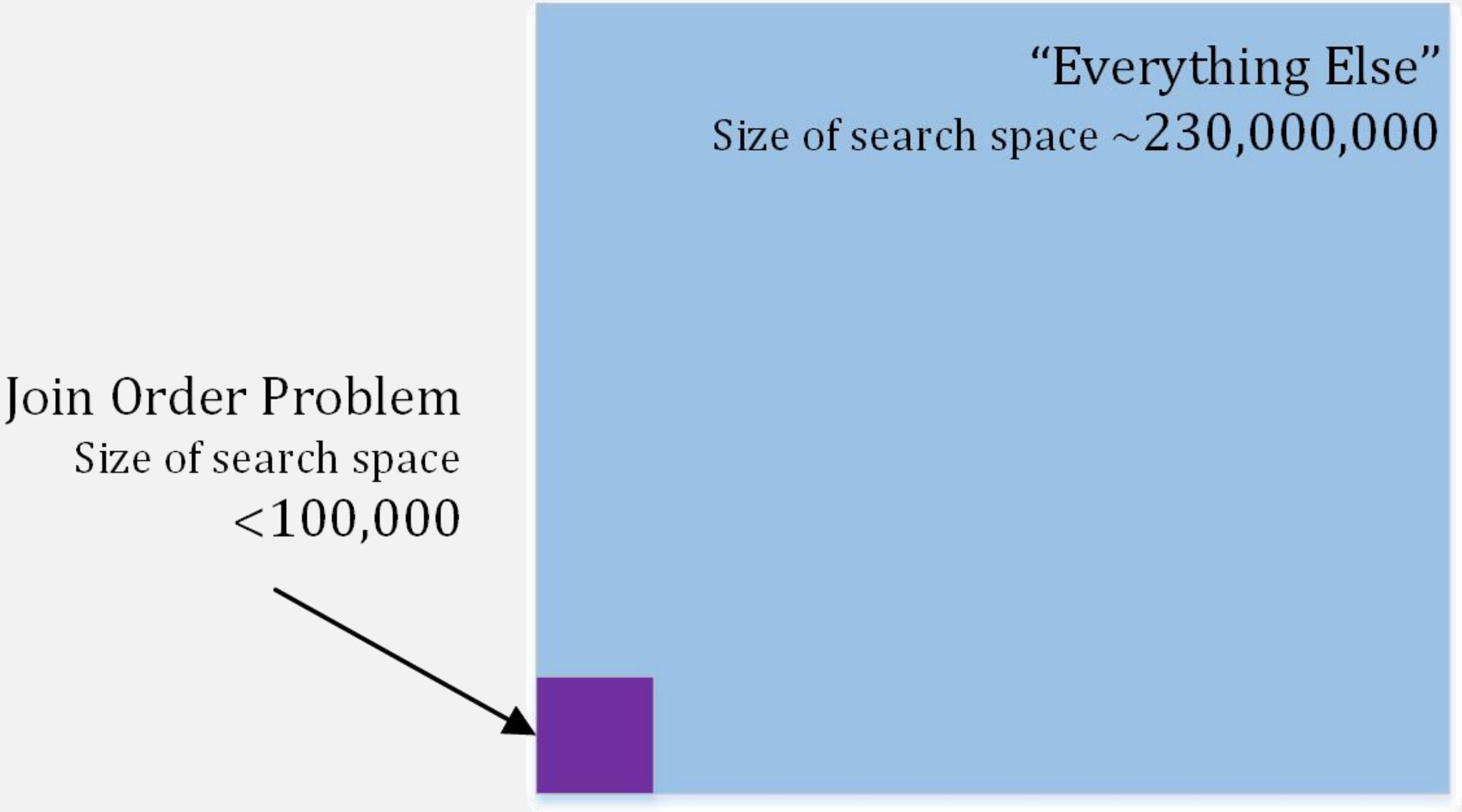
- Massively Parallel Processing(OLAP eg. Teradata ,SQL Server PDW,SAP HANA,Vertica ...)
  - Legacy Planner was not initially designed with **distributed data** processing in mind
- SQL on Hadoop
  - Hive,Stinger (MapReduce)
  - Impala,HAWQ,Presto (specialized query engines without MapReduce)

Textbook Query Optimizer

- A classical two stage optimization(STRATIFIED):
  - Logical Query Optimization (Rule Oriented)
  - Physical Query Optimization (Cost/Hint Oriented)
- Addresses Join re-ordering
- Treats everything else as “add-on” (grouping, with clause, etc.)
- Imposes order on specific optimization steps
- Recursively descends into sub-queries

Join Ordering vs. “Everything Else”

- TPC-H Query 5
  - 6 Tables
  - “Harmless” query



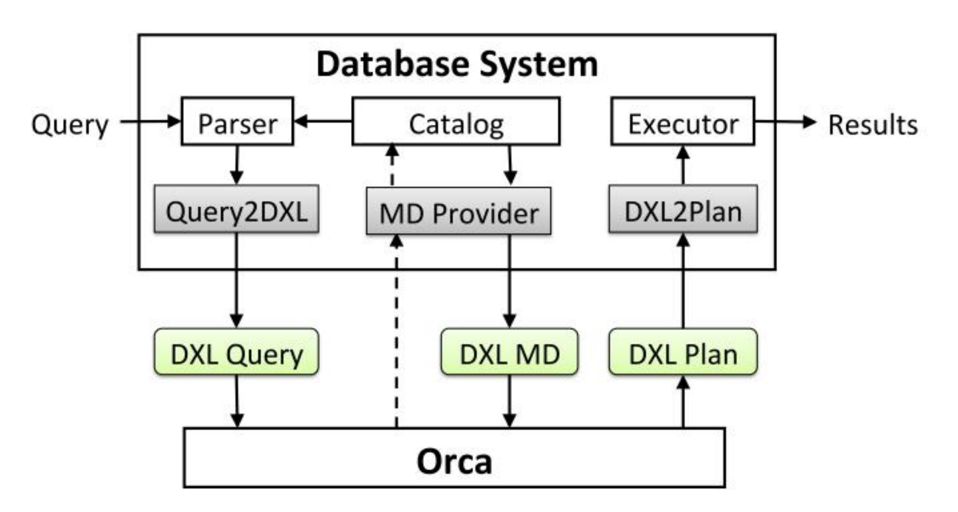
ORCA

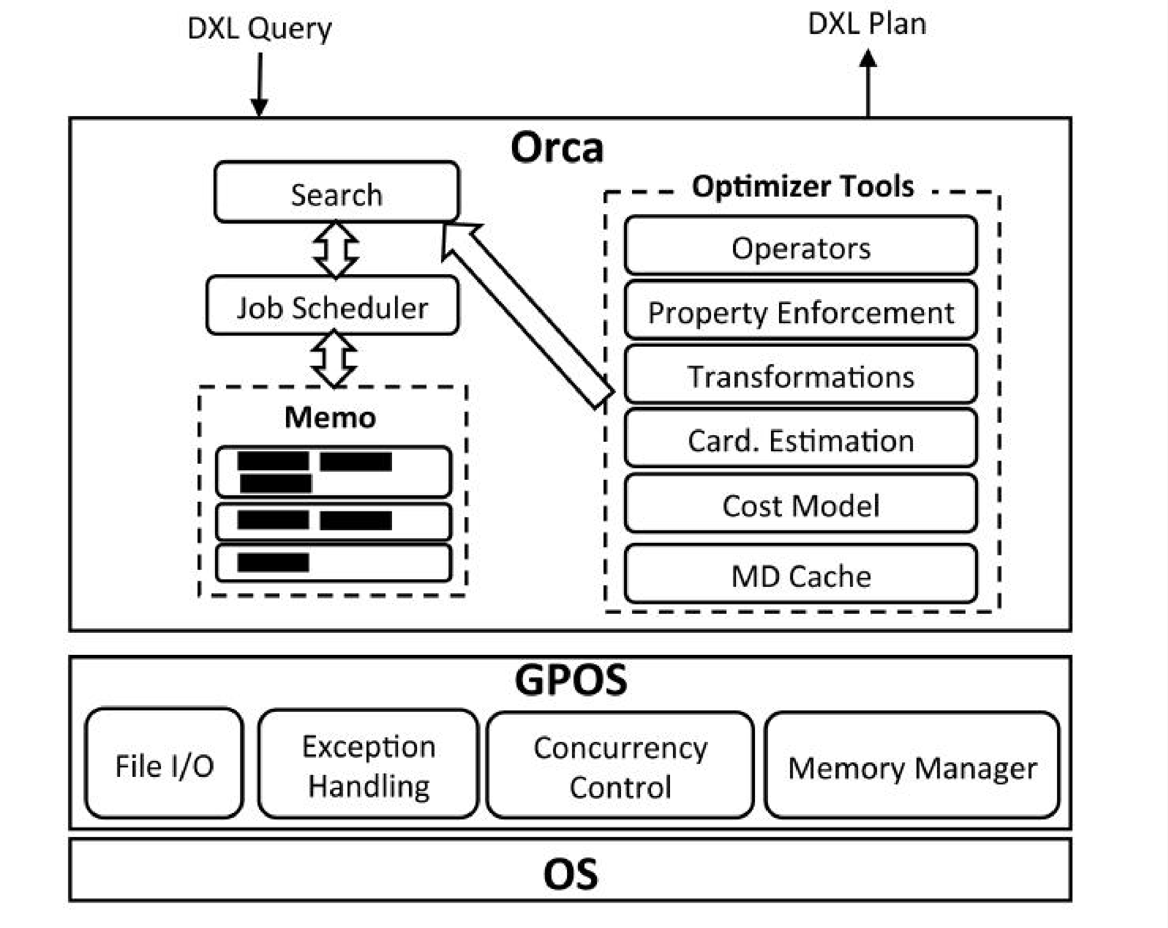
Orca is a modern top-down query optimizer based on the **Cascades** optimization framework.

- Modularity
  - on longer confined to a specific host system like traditional optimizers
- Extensibility
  - on longer multi-phase optimization
- Multi-core ready
  - speed-up of the optimization process
- Verifiability
- Performance

Orca architecture

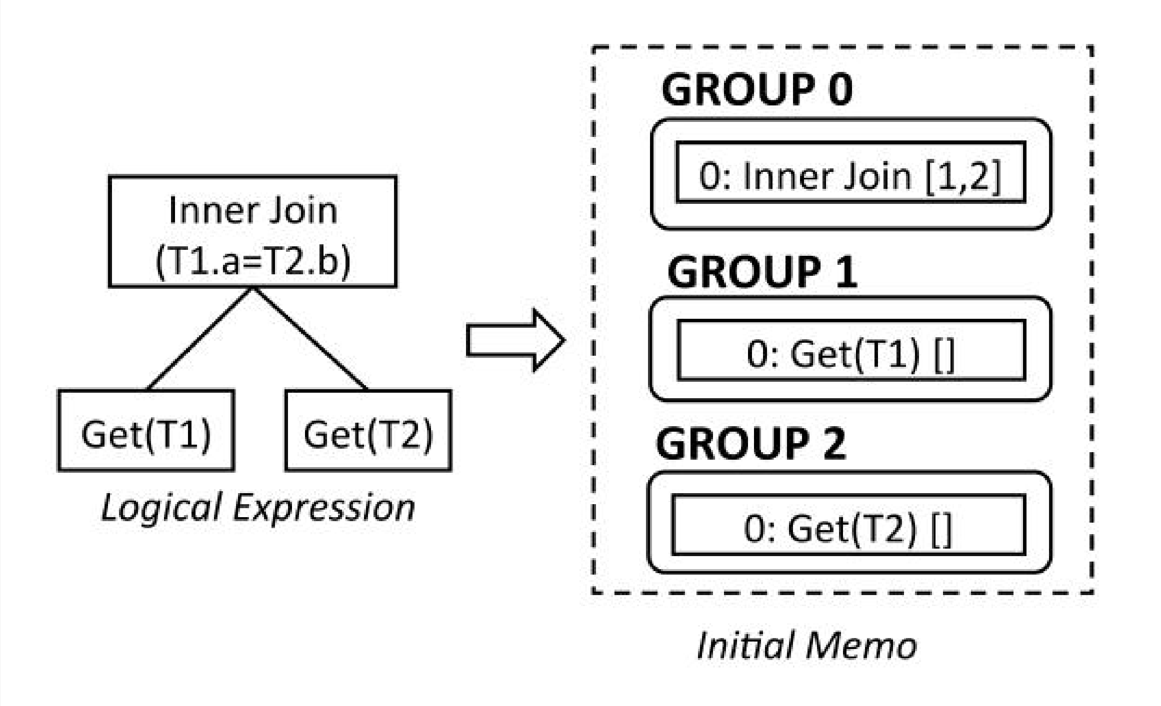
- Decoupling the optimizer from the database system
  - Data eXchange Language (DXL)





Memo Table

- Group
  - Container of equivalent expressions
- GroupExpression
  - operator that has other groups as its children
- Transformation
  - Exploration
  - Implementation



Copying-in initial logical expression

WorkFlow

- Exploration
- Stats Derivation
- Implementation
- Optimization

- Search and Job Scheduler
  - Exploration,Implementation,Optimization

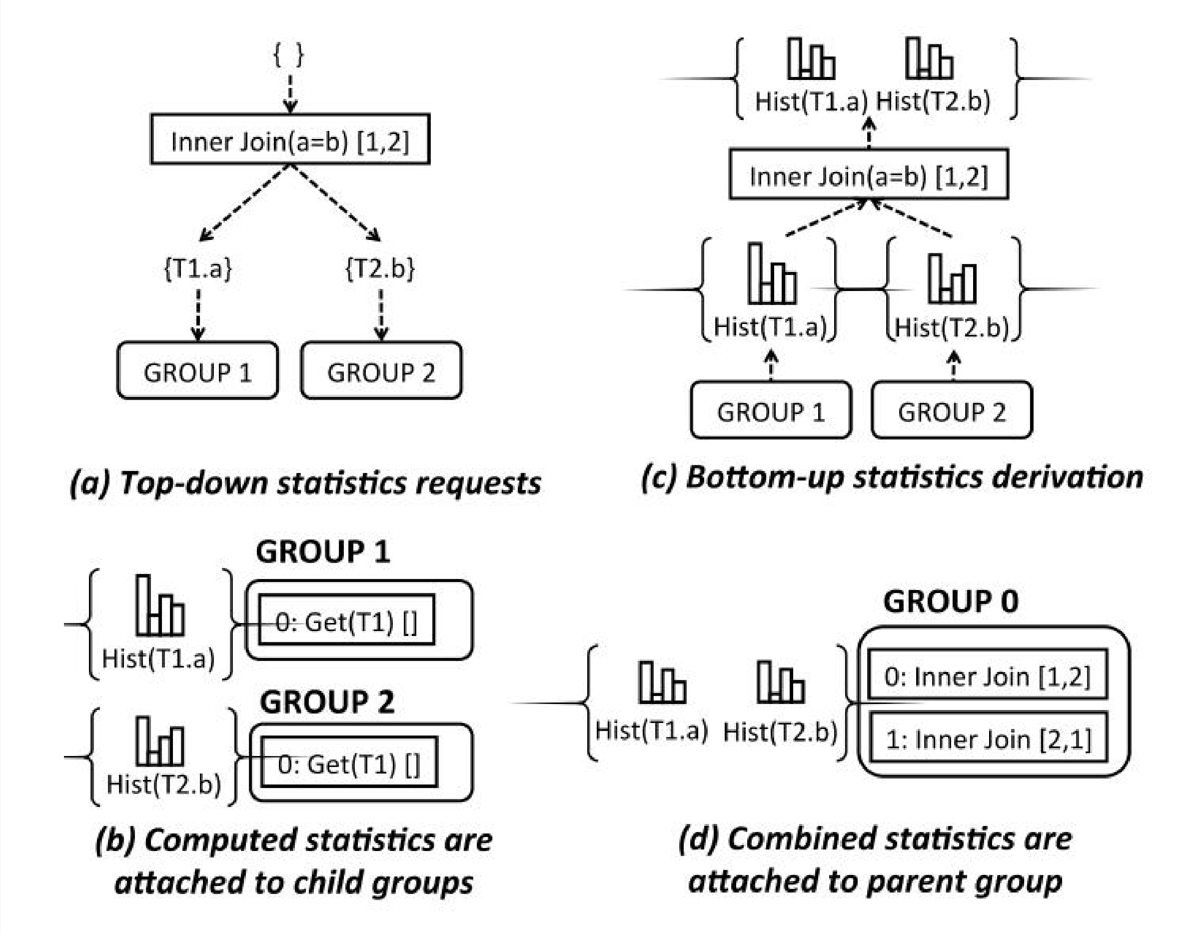
- Example:
  - SELECT T1.a FROM T1, T2 WHERE T1.a = T2.b ORDER BY T1.a;
  - Hashed(T1.a), Hashed(T2.a)

Exploration

- Explorable Transformation
  - Join Reorder (naive)
  - Decorrelation
  - Predicate Push Down
  - Eager Aggregate
- Transformation Pattern
- Explore Children Before Explore Self

Stats Derivation

- We need to propagate statistics
- Choose a group expression with highest promise



Implementation

- Generate all physical implementation for all logical operators
  - Transformation rules that create physical implementations of logical expressions are triggered(Get2Scan rules,InnerJoin2HashJoin and InnerJoin2NLJoin rules)

Optimization

- Enforce distribution and ordering requirements and pick the cheapest plan
- Optimization starts by submitting an initial optimization request to the Memo's root group

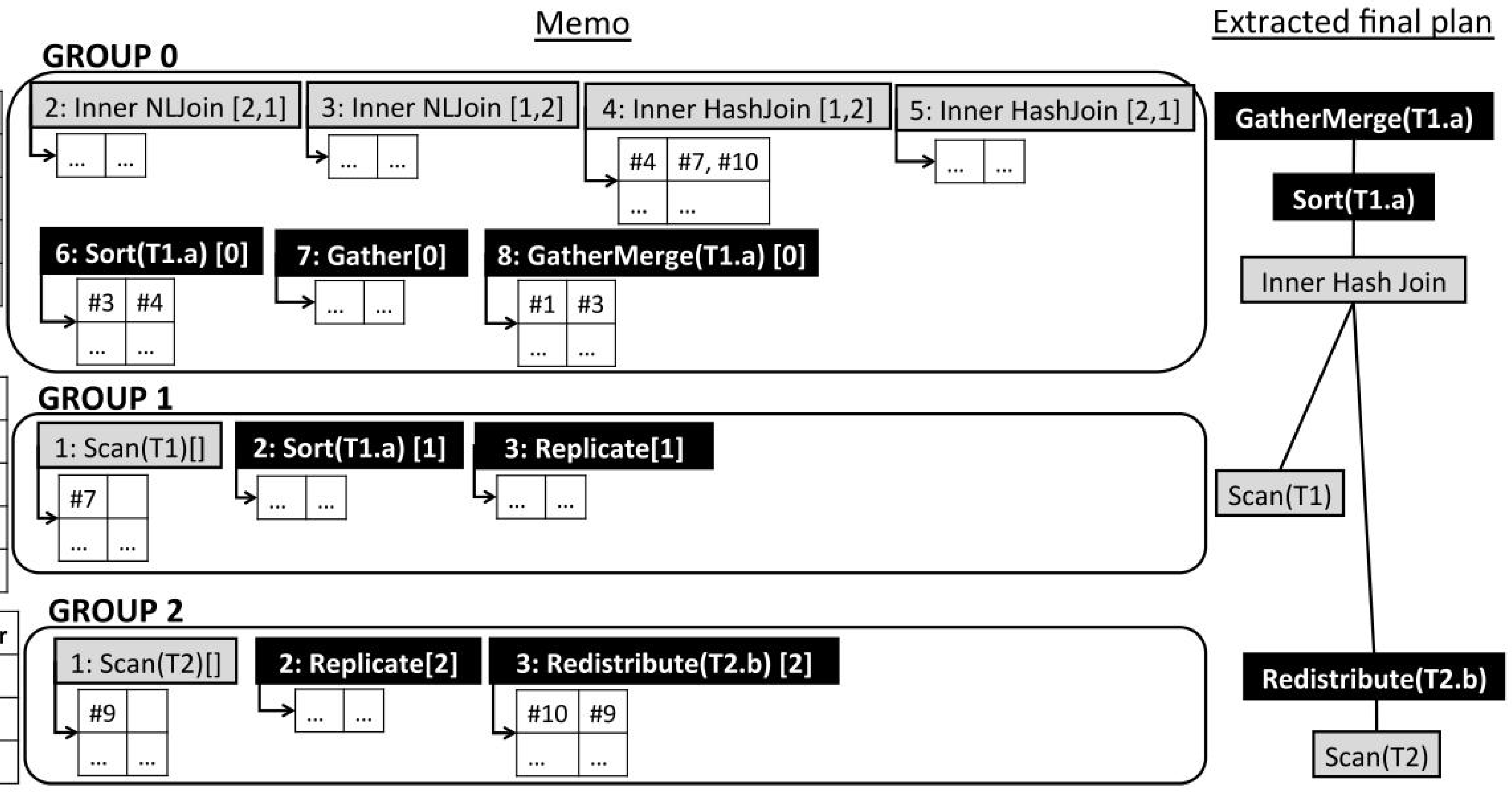
A Running Example

Groups Hash Tables

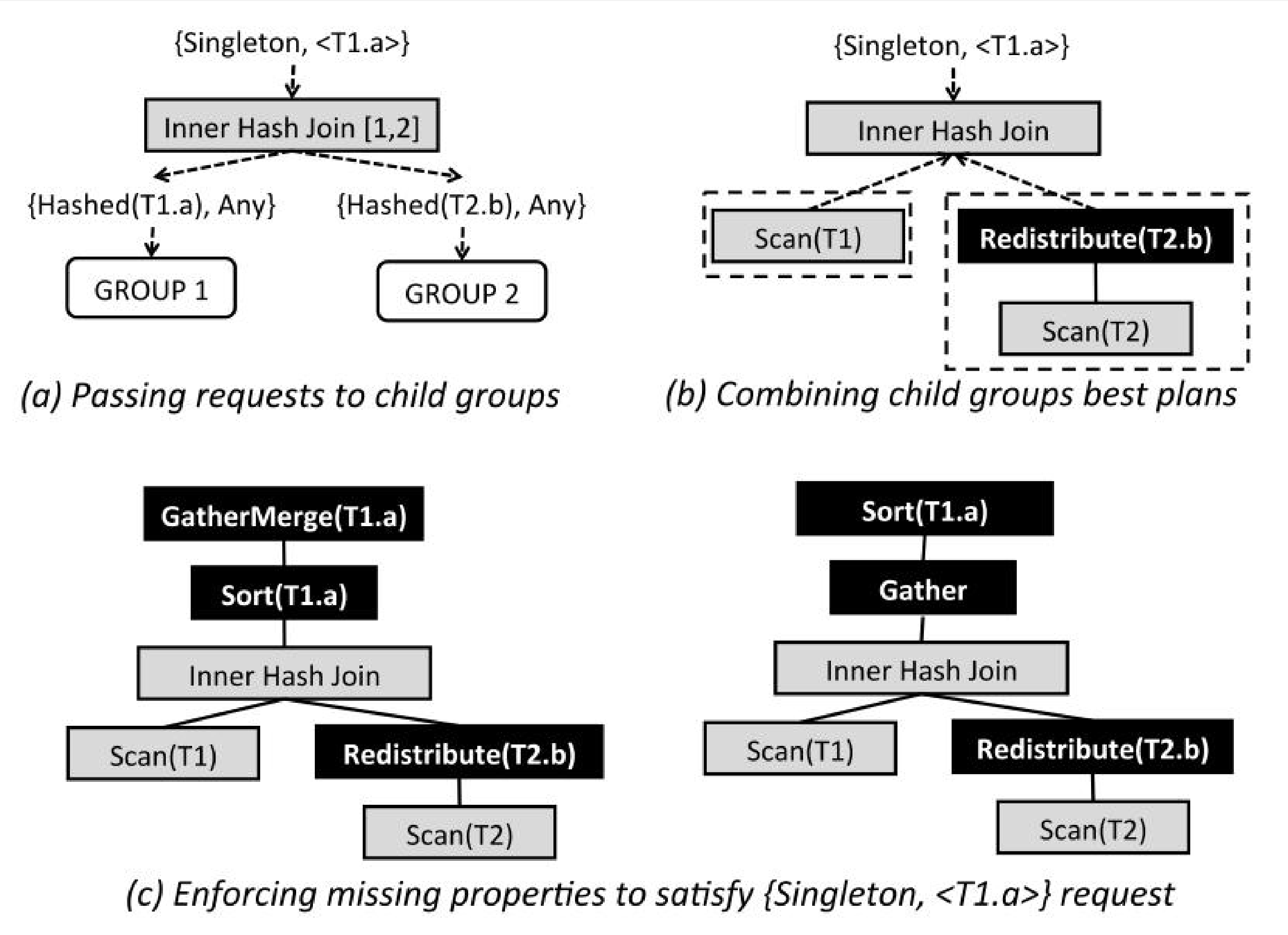
#	Opt. Request	Best GExpr
1	Singleton, <T1.a>	*8*
2	Singleton, Any	*7*
3	Any, <T1.a>	*6*
4	Any, Any	*4*

#	Opt. Request	Best GExpr
5	Any, Any	*1*
6	Replicated, Any	*3*
7	Hashed(T1.a), Any	*1*
8	Any, <T1.a>	*2*

#	Opt. Request	Best GExpr
9	Any, Any	*1*
10	Hashed(T2.b), Any	*3*
11	Replicated, Any	*2*



A Running Example



(a) Passing requests to child groups

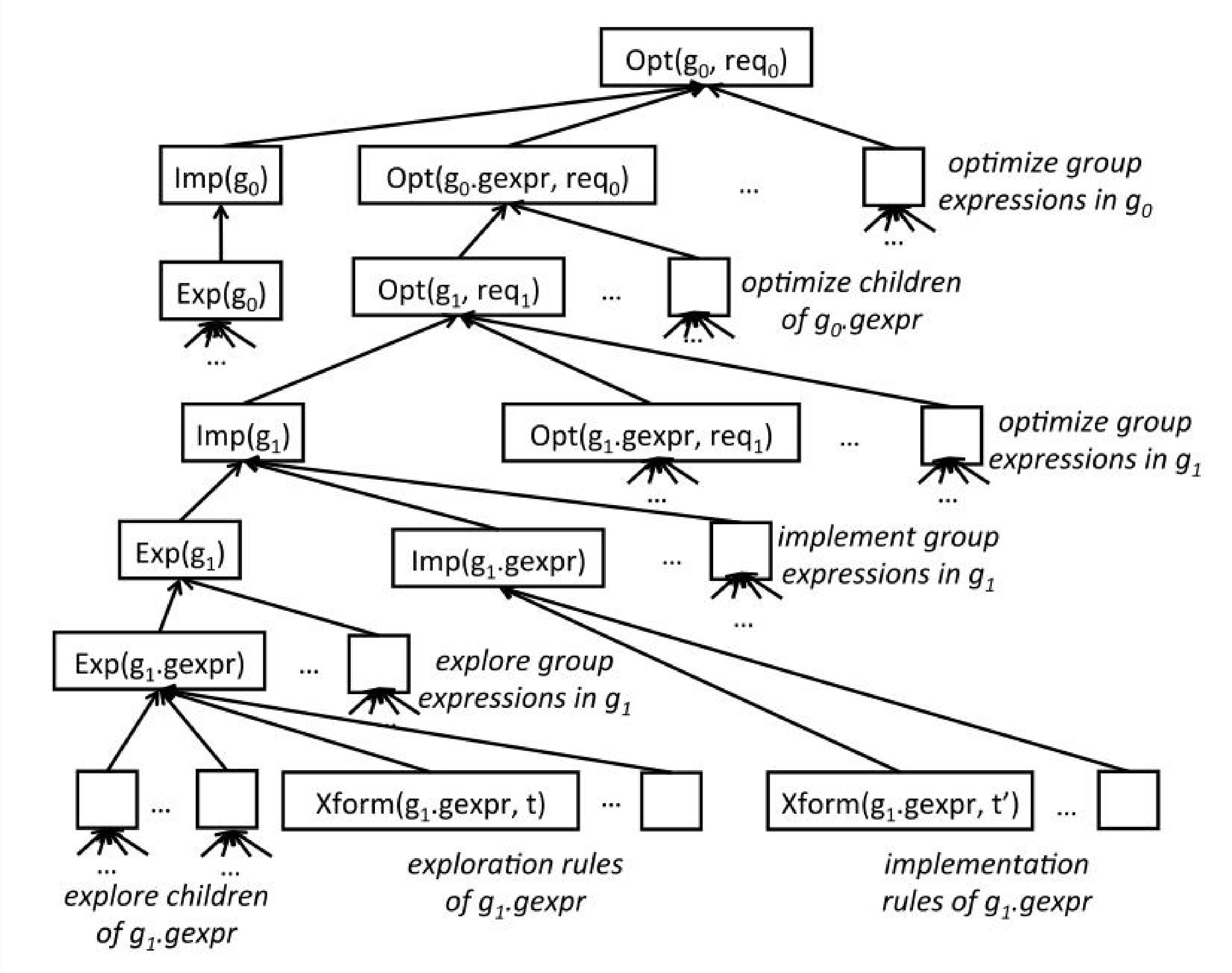
(b) Combining child groups best plans

(c) Enforcing missing properties to satisfy {Singleton, <T1.a>} request

Parallel Query Execution

- Optimization work is broken to small work units called jobs.
  - Exp(g)
  - Exp(gexpr)
  - Imp(g)
  - Imp(gexpr)
  - Opt(g)
  - Opt(gexpr)
  - Xform(gexpr, t)

Parallel Query Execution



Verifiability & Performance

- Verifiability
  - AMPERe is a tool for Automatic capture of Minimal Portable and Executable Repros
  - TAQO for Testing the Accuracy of Query Optimizer

- Performance
- Average time to fix customer issues：
  - Legacy Optimizer (Planner) ~ **70** days
  - Pivotal Query Optimizer (Orca) ~ **13** days

Part II:

The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database

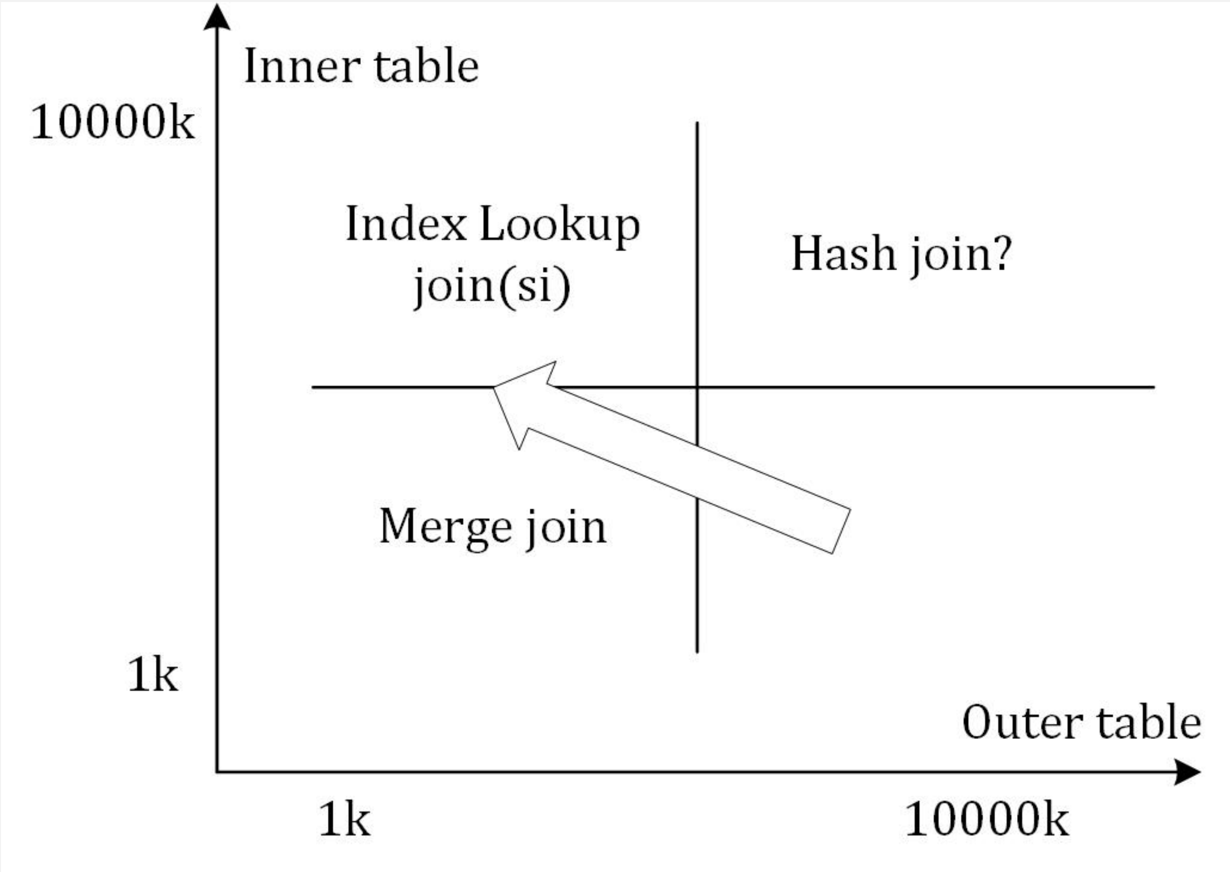
Motivation

- Enterprises need to run **complex analytic queries** on real-time for interactive real-time decision making
- Analytical queries need to be optimized and executed very **quickly**

MEMSQL

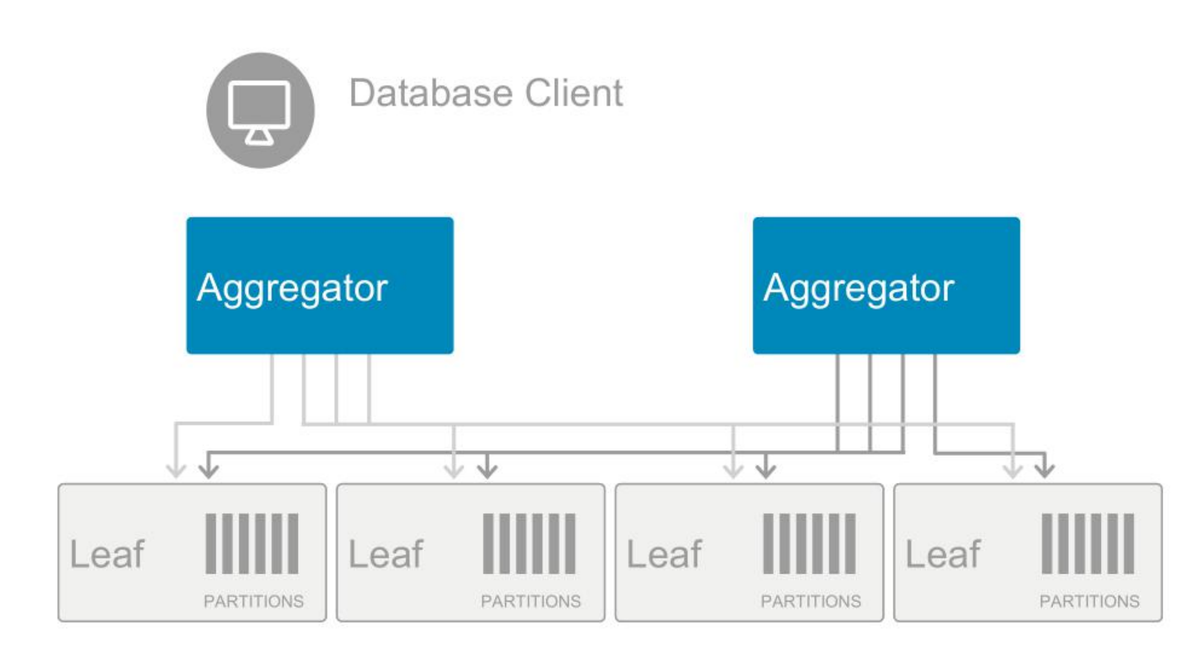
- Is a distributed **memory-optimized** SQL database
- Real-time transactional and analytical workloads
- Can store data in two formats:
  - in-memory row-oriented
  - disk-backed column-oriented
- **Sub-second query latencies** over large volumes of changing data

Big Data Dilemma



MEMSQL Architecture

- **Shared-nothing** architecture
- Two types of nodes:
  - Aggregator nodes = scheduler nodes
  - Leaf nodes = execution nodes
- Two ways to distribute the user data based on table
  - Distributed tables - rows are **sharded** across the leaf nodes
  - Reference tables - the table data is **replicated** across all nodes



MemSQL: Execution of a query

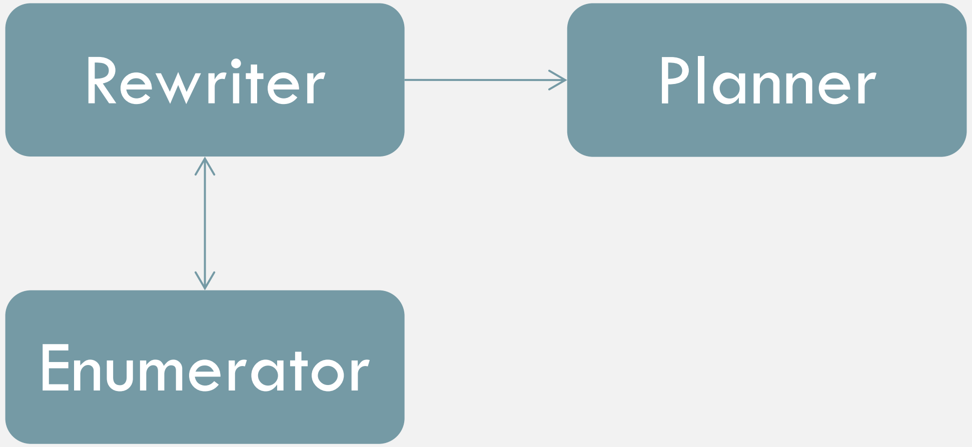
- Aggregator node: Converts the query into a distributed query execution plan – DQEP
- Series of DQEPs = operations which are executed on nodes
- Representation of DQEPs using a SQL-like syntax and interface
- Query plans are compiled to machine code and are cached, without values for the parameters

Components of the Optimizer

- Rewriter
  - Applies SQL-to-SQL rewrites on the query, using heuristics or cost (based on the characteristics of the query and the rewrite itself)
  - Applies some rewrites in a top-down manner, while applying others in a bottom-up manner and interleaves rewrites
- Enumerator
  - Determines the distributed join order and data movement decisions
  - Selects the best plan, **based on the cost models** of the database operations and the network data movement operations
  - Called by the Rewriter to cost rewrites

Components of the Optimizer

- Planner
  - Converts the logical execution plan to a sequence of distributed query and data movement operations
  - Uses SQL extensions: RemoteTables and ResultTables



Rewriter: Heuristic Rewrites

- Column Elimination transformation: removes any projection columns that are never used
  - reduce I/O cost and network resources
- Sub-Query Merging: Merges subselects
  - Disadvantage:
  - In the case of joining very large numbers of tables under a number of simple views, merging all the subselects would result in a single large join of all these tables
  - discards information about the structure of the join graph & expensive for the Enumerator to effectively optimize
  - Solution: Uses heuristics to detect this type of situation and avoid merging all the views in such cases
- Sub-Query convert to join

Rewriter: Cost-Based Rewrites

- Group-By Pushdown: reorders a 'group by' before a join to evaluate the group by earlier

- This transformation is **not always beneficial,** depending on the sizes of the joins and the cardinality of the group by
  - needing of cost estimates

Interleaving of Rewrites

- Pushing a predicate down may enable Outer Join to Inner Join conversion if that predicate rejects NULLs of the outer table
- Interleaving of two rewrites: going **top-down** over each select block (before processing any subselects) and apply
  - 1) Outer Join to Inner Join and then
  - 2) Predicate Pushdown

- Rewrites like bushy join are done **bottom-up,** because they are cost-based

Costing Rewrites

- CREATE TABLE T1 (a int, b int, shard key (b)) CREATE TABLE T2 (a int, b int, shard key (a), unique key (a))
  - Q1: SELECT sum(T1.b) AS s FROM T1, T2
    WHERE T1.a = T2.a
    GROUP BY T1.a, T1.b
  - Q2: SELECT V.s from T2,
    (SELECT a, sum(b) as s
    FROM T1
    GROUP BY T1.a, T1.b
) V
    WHERE V.a = T2.a;

R1=200,000 be the rowcount of T1 and R2=50,000

lookup cost of Cj=1 units,the group-by is executed using a hash table with an average cost of Cg=1 units per row

Sg= 1/4 be the fraction of rows of T1 left after grouping on (T1.a, T1.b),Sj= 1/10 be the fraction of rows of T1 left after the join between T1.a and T2.a

Cost Q1=R1*Cj+R1*Sj*Cg=

200,000*Cj+20,000*Cg=220,000

Cost Q2=R1*Cg+R1*Sg*Cj=

200,000*Cg+50,000*Cj=250,000

Costing Rewrites

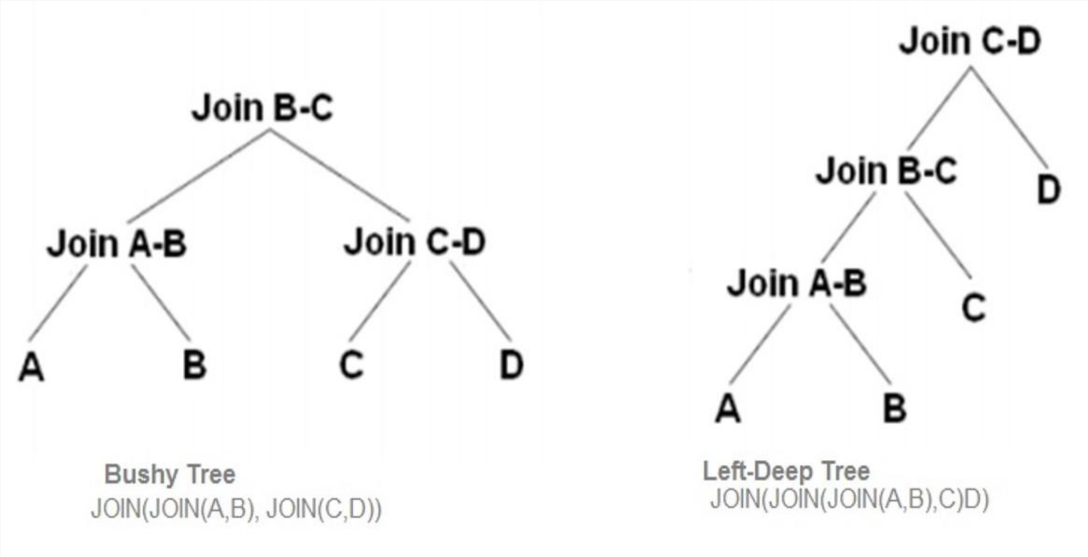
- Run the query in a **distributed** setting
- T2 is sharded on T2.a, but T1 is not sharded on T1.a → *compute this join by reshuffling*
- $Cr = 3$ units per row
- Cost Q1=R1*Cr +R1*Cj+R1*Sj*Cg=

  200,000*(Cr+Cj)+20,000*Cg=620,000
- Cost Q2=R1*Cg+R1*Sg*Cr+R1*Sg*Cj=

  200,000*Cg+50,000*(Cr+Cj)=400,000
- clusters with slower network where **network costs** may often dominate the cost of a query

Bushy Joins

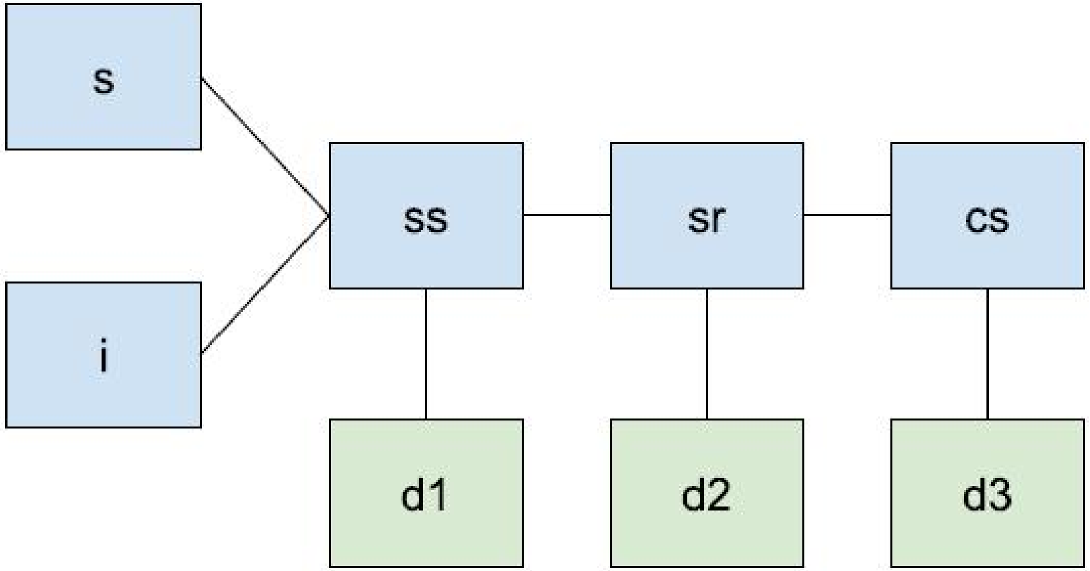
- Finding the optimal join permutation extremely costly and time consuming
- Many database systems do not consider bushy joins limiting their search join trees
- **Query rewrite mechanism to generate bushy join plans** is not new and has already been explored
- MemSQL use Bushy joins plan
- Use **heuristic** – based approach which consider only hopeful bushy joins

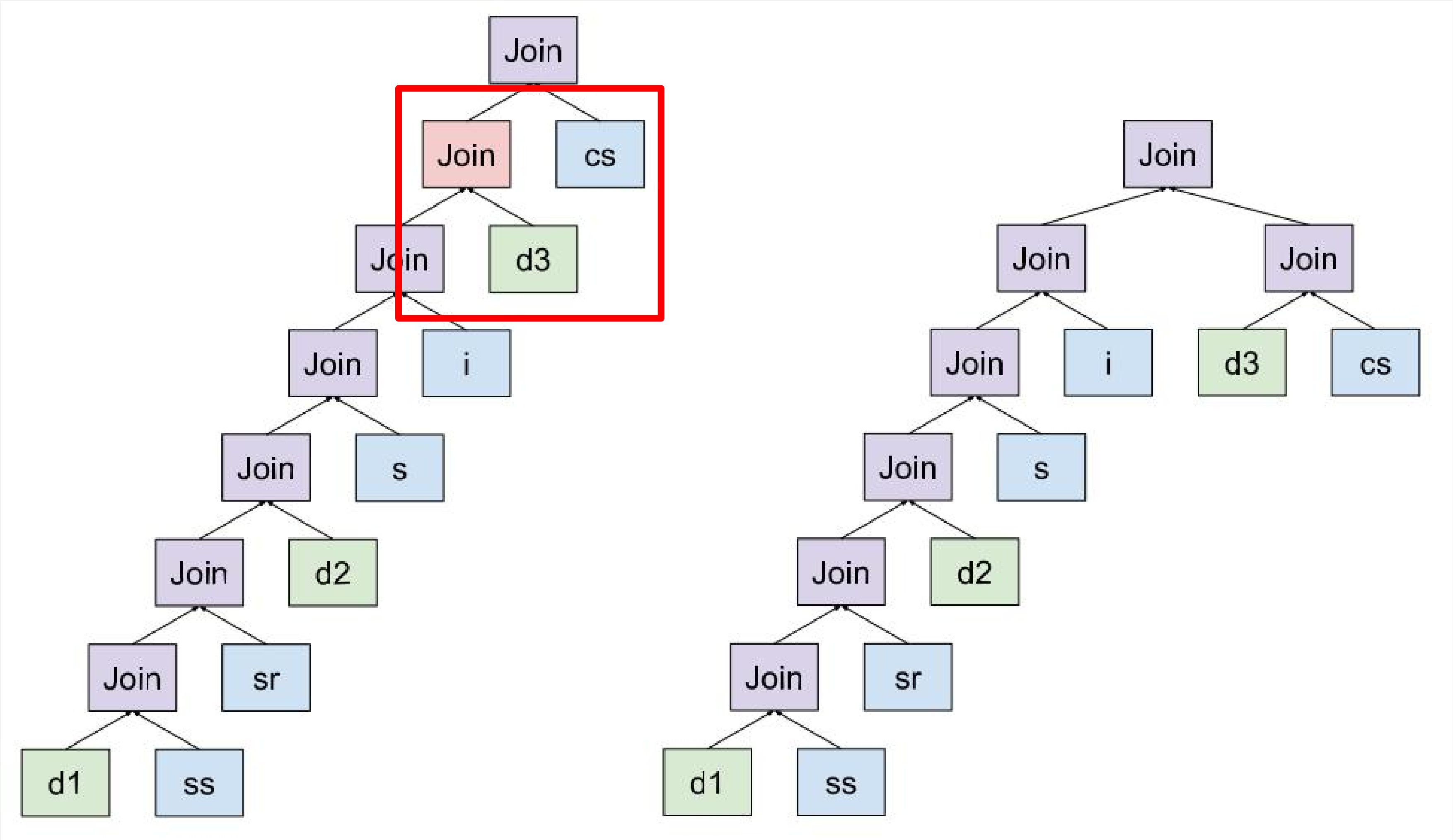


Generate bushy plans - Algorithm

1. **Build a graph** where vertexes represent tables and edges represent join predicate
2. Identify **candidate *satellite* tables**
3. **Select only the *satellite* tables,** which are the tables connected to only other table in the graph
4. Identify ***seed* tables,** which are tables that are connected to at least **two different tables, at least** one of which is **a *satellite* table.**
5. For each *seed* table:
   a. Compute the **cost C1** of the current plan
   b. Create **a derived table** containing the *seed* table joined to its adjacent *satellite* tables
   c. **Apply the *Predicate Pushdown*** rewrite followed by the ***Column Elimination*** *rewrite*
   d. Compute the **cost C2.** If C1 < C2, discard the changes made in steps (b) and (c), and otherwise keep them.

```
SELECT …
FROM   store_sales ss,
       store_returns sr,
       catalog_sales cs,
       date_dim d1,
       date_dim d2,
       date_dim d3,
       store s,
       item i
WHERE  d1.d_moy = 4
       AND d1.d_year = 2000
       AND d1.d_date_sk = ss_sold_date_sk
       AND i_item_sk = ss_item_sk
       AND s_store_sk = ss_store_sk
       AND ss_customer_sk = sr_customer_sk
       AND ss_item_sk = sr_item_sk
       AND ss_ticket_number = sr_ticket_number
       AND sr_returned_date_sk = d2.d_date_sk
       AND d2.d_moy BETWEEN 4 AND 10
       AND d2.d_year = 2000
       AND sr_customer_sk = cs_bill_customer_sk
       AND sr_item_sk = cs_item_sk
       AND cs_sold_date_sk = d3.d_date_sk
       AND d3.d_moy BETWEEN 4 AND 10
       AND d3.d_year = 2000
GROUP BY …
ORDER BY …
```




Join
Join
cs
Join
d3
Join
i
Join
s
Join
d2
Join
sr
d1
ss
Join
Join
Join
Join
i
d3
cs
Join
s
Join
d2
Join
sr
d1
ss

```sql
SELECT …
FROM   store_sales ss,
       store_returns sr,
       catalog_sales cs,
       date_dim d1,
       date_dim d2,
       date_dim d3,
       store s,
       item i
WHERE  d1.d_moy = 4
       AND d1.d_year = 2000
       AND d1.d_date_sk = ss_sold_date_sk
       AND i_item_sk = ss_item_sk
       AND s_store_sk = ss_store_sk
       AND ss_customer_sk = sr_customer_sk
       AND ss_item_sk = sr_item_sk
       AND ss_ticket_number = sr_ticket_number
       AND sr_returned_date_sk = d2.d_date_sk
       AND d2.d_moy BETWEEN 4 AND 10
       AND d2.d_year = 2000
       AND sr_customer_sk = cs_bill_customer_sk
       AND sr_item_sk = cs_item_sk
       AND cs_sold_date_sk = d3.d_date_sk
       AND d3.d_moy BETWEEN 4 AND 10
       AND d3.d_year = 2000
GROUP BY …
ORDER BY …
```

```sql
SELECT …
FROM   store_sales,
       store_returns,
       date_dim d1,
       date_dim d2,
       store,
       item,
       (SELECT *
        FROM   catalog_sales,
               date_dim d3
        WHERE  cs_sold_date_sk = d3.d_date_sk
               AND d3.d_moy BETWEEN 4 AND 10
               AND d3.d_year = 2000) sub
WHERE  d1.d_moy = 4
       AND d1.d_year = 2000
       AND d1.d_date_sk = ss_sold_date_sk
       AND i_item_sk = ss_item_sk
       AND s_store_sk = ss_store_sk
       AND ss_customer_sk = sr_customer_sk
       AND ss_item_sk = sr_item_sk
       AND ss_ticket_number = sr_ticket_number
       AND sr_returned_date_sk = d2.d_date_sk
       AND d2.d_moy BETWEEN 4 AND 10
       AND d2.d_year = 2000
       AND sr_customer_sk = cs_bill_customer_sk
       AND sr_item_sk = cs_item_sk
```

Enumerator

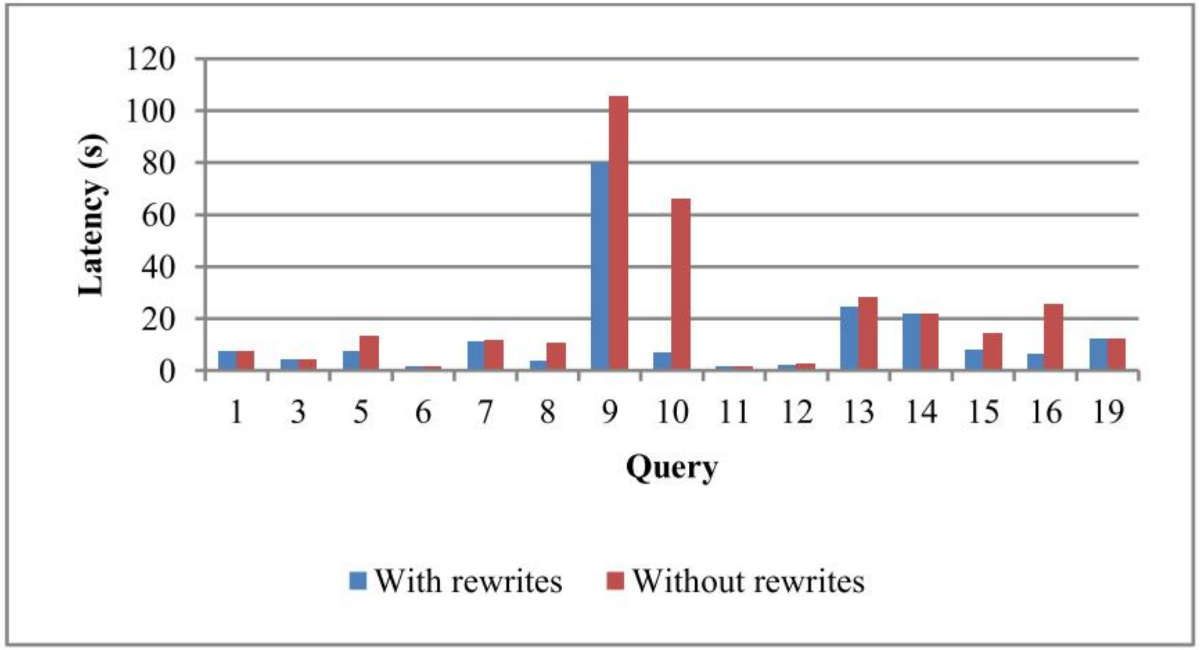
- **Prune** heavily to eliminate a huge majority of the search space
- Enumerator uses several heuristics to generate initial candidate join orders
  - Cost each candidate join order
  - Cheapest candidate provides an **initial upper bound** on the cost
- Details are in the paper
  - Query optimization time: The new bottleneck in real-time analytics.

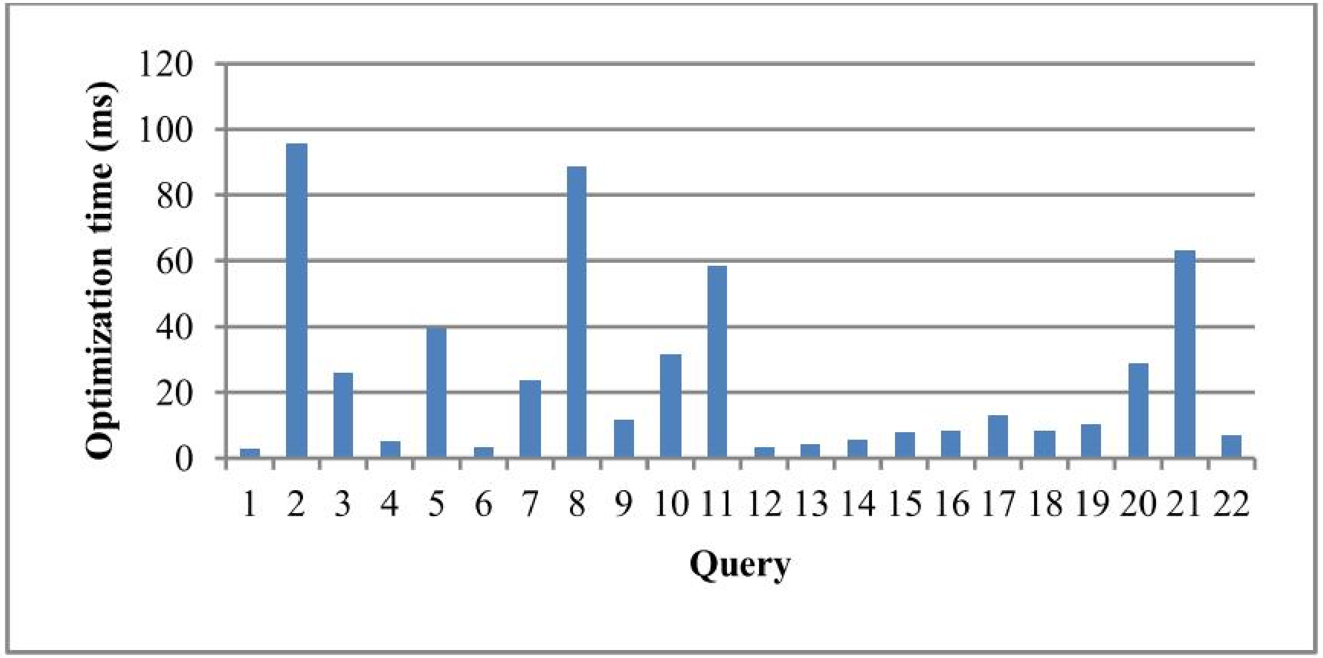
Planner: Remote Tables and Result Tables

- Remote Tables
  - Communication between each leaf and all the partitions
  - Problem: Each partition querying all other partitions

- Result Tables (SQL SELECT)
  - Store intermediate results for each partition and then compute the final select

Experiments





TPC-H at Scale Factor 100

Experiments

Query	Tables	Pruning %
Q3	3	25.00%
Q5	6	61.46%
Q7	6	72.92%
Q8	8	95.80%
Q9	6	84.90%
Q21	6	62.50%

Pruning percentage huge for most queries

Experiments

Query	Optimization Overhead	Execution Speedup
Q15	13%	5.8x
Q25	16%	10.1x
Q46	12%	2.85x

bushy join significant execution speedup with minimal optimization overhead

ORCA vs. MEMSQL

	ORCA	MEMSQL
Scalability	modular,support different architecture	only for memsql
Plan space	enormous	limit
Optimization time	seconds level	<100ms
Utilization technology	multicore	heuristics,schema

References

- Graefe G. **The Cascades Framework for Query Optimization**[J]. Data Engineering Bulletin, 1995, 18(5):19-29.
- R. Sen, J. Chen, and N. Jimsheleishvilli. **Query optimization time: The new bottleneck in real-time analytics**. In Proceedings of the 3rd VLDB Workshop on In-Memory Data Management and Analytics, page 8. ACM, 2015.
- Bacon, D. F., Cooper, B. F., Kogan, E., & Woodford, D. (2017). **Spanner : Becoming a SQL System**, 331–343.
- Shute, J., Vingralek, R., Samwel, B., & Rae, I. (2013). **F1: A distributed SQL database that scales**. Proceedings of the …, 6(11), 1068–1079.

End,thanks!