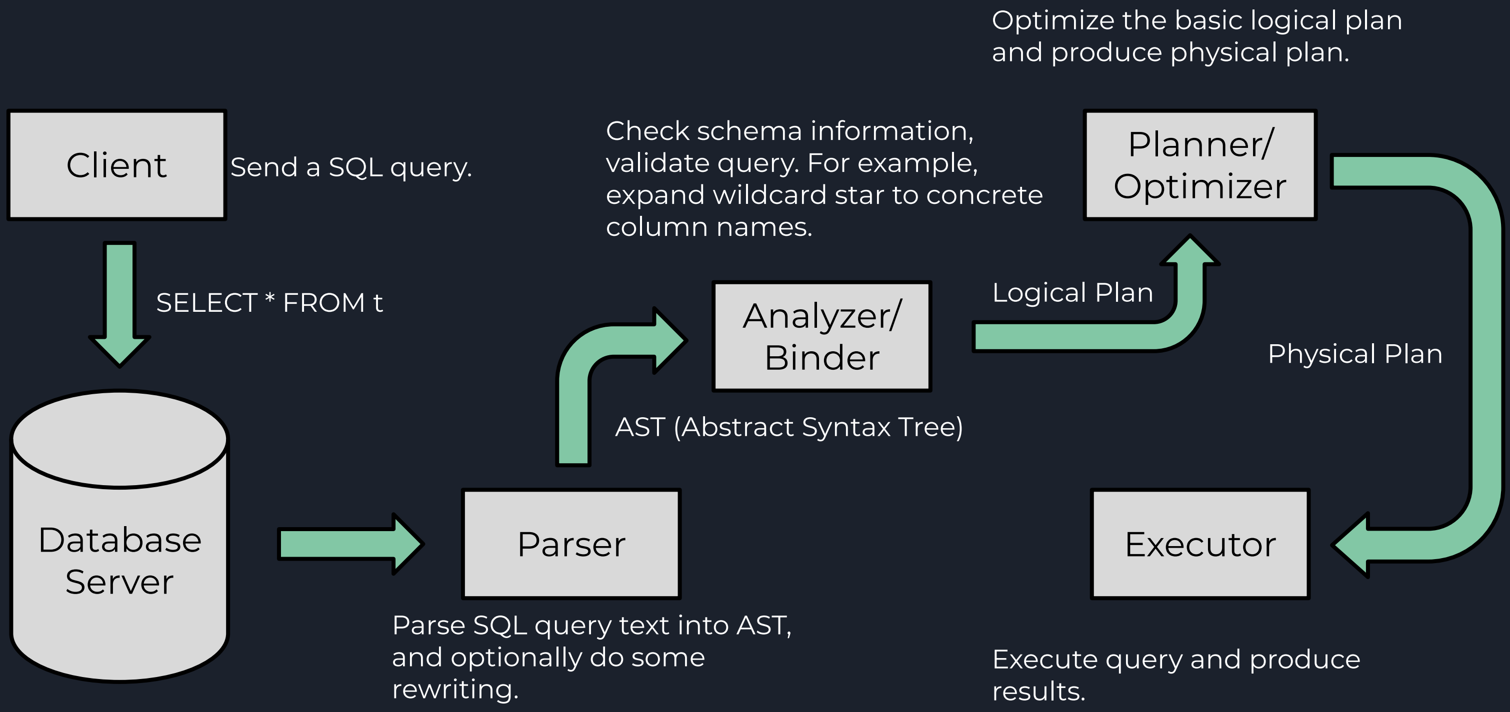
SQL processing & query optimization

Presented by leiysky

Schedule

- Life of a SQL query
- SQL parser & binder
- Query optimizer

Life of a SQL query


Optimize the basic logical plan
and produce physical plan.
Client
Send a SQL query.
Check schema information,
validate query. For example,
expand wildcard star to concrete
column names.
Planner/
Optimizer
SELECT * FROM t
Logical Plan
Analyzer/
Binder
Physical Plan
AST (Abstract Syntax Tree)
Database
Server
Parser
Executor
Parse SQL query text into AST,
and optionally do some
rewriting.
Execute query and produce
results.

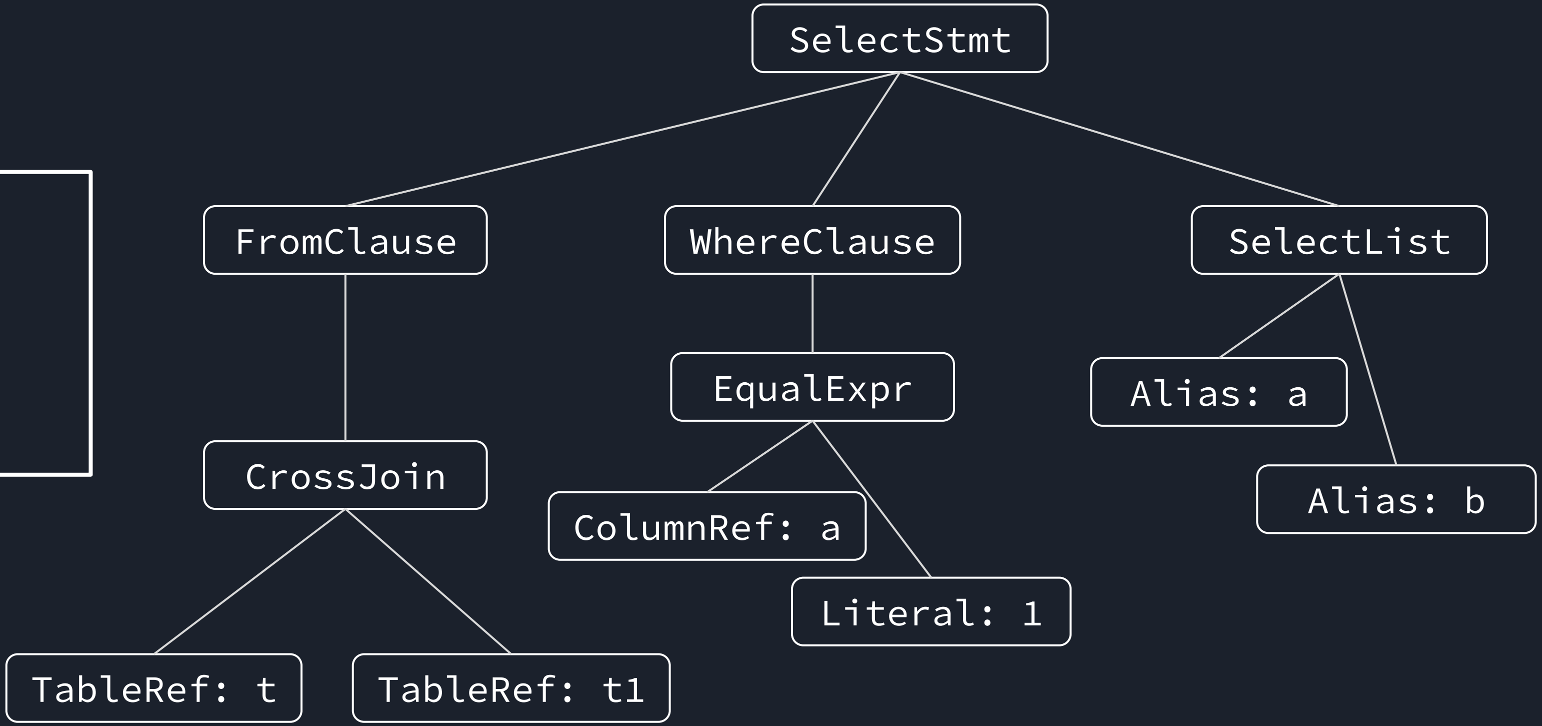
SQL Parser & Binder

SQL grammar

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES } ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ] [...] ]
```

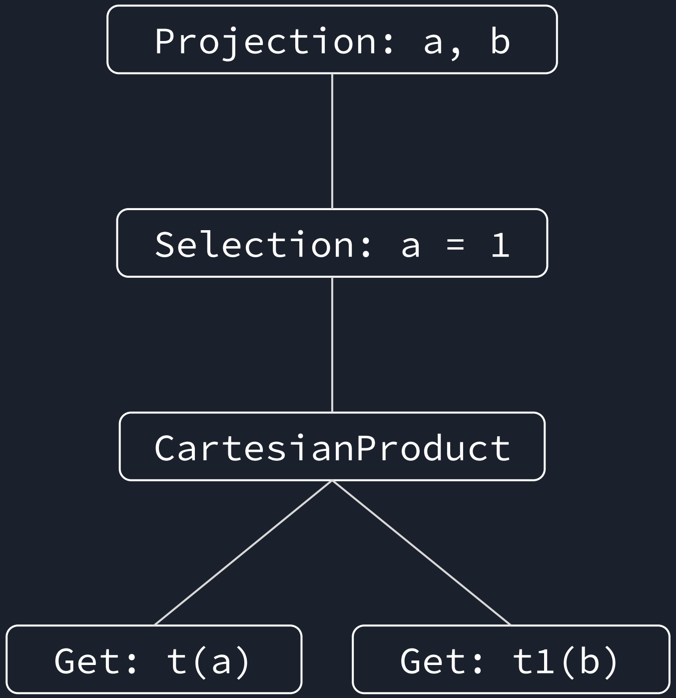
SQL AST

```sql
SELECT a, b

FROM t, t1

WHERE a = 1
```

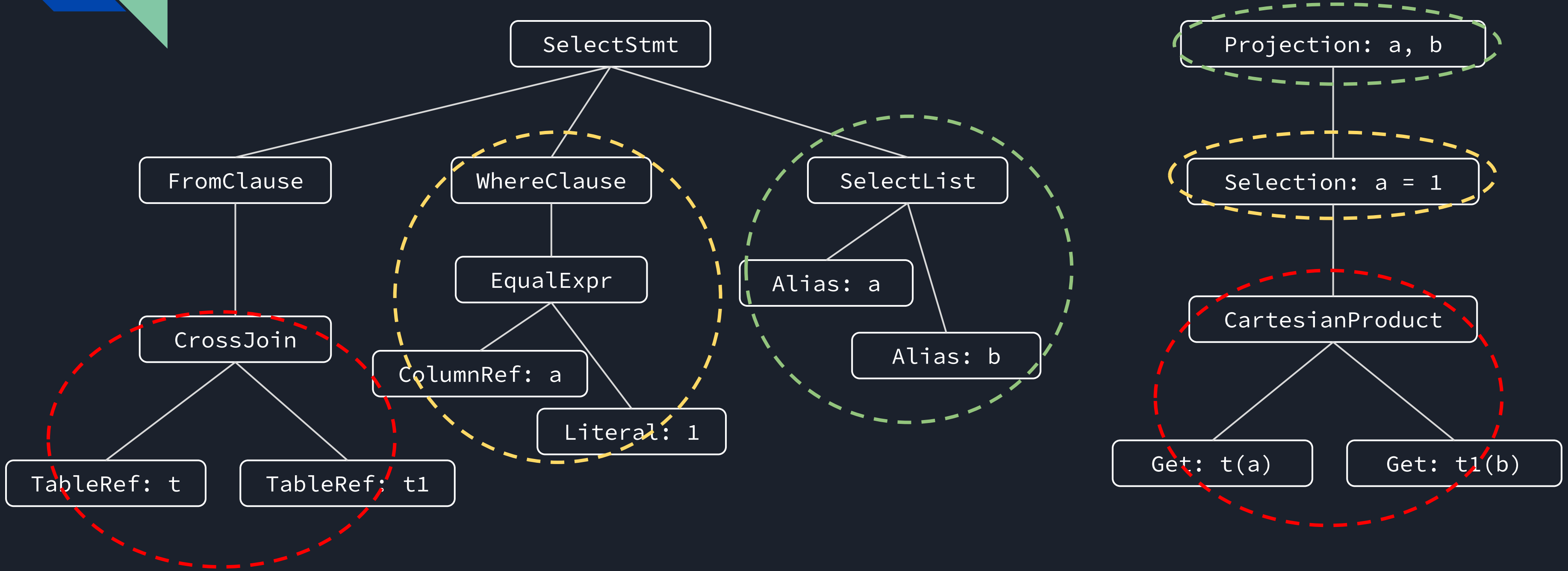


Relational Algebra

```
SELECT a, b
FROM t, t1
WHERE a = 1
```



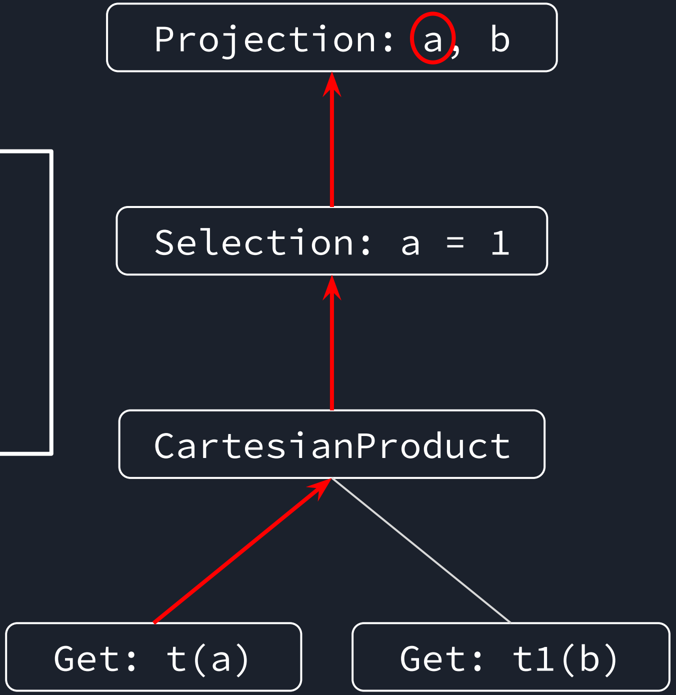
Mapping from AST to Algebra



Name resolution

- Who Am I?
- Where Am I?
- Where Am I Going?

```
SELECT a, b
FROM t, t1
WHERE a = 1
```



Name resolution

- A more complicated case
- Q1: What is output "a"?
- Q2: Which table does "a" come from?

- A1: 2
- A2: shown in figure

```
CREATE TABLE t(a INT, b INT)

INSERT INTO t VALUES(1, 2)
```

```
SELECT a
FROM (
  SELECT * FROM (
    SELECT a AS b, b AS a FROM t
  ) AS t1(a)
  CROSS JOIN t AS t2(b)
) AS t(b, a)
```

Name resolution

```
SELECT a
FROM (
  SELECT * FROM (
    SELECT a AS b, b AS a FROM t
  ) AS t1(a)
  CROSS JOIN t AS t2(b)
) AS t(b, a)
```

```
Ctx 3: t=(b, a, b, b) result=(a)
```

```
Ctx 2: t1=(a, a) t2=(b, b)
result=(a, a, b, b)
```

```
Ctx 1: t=(a, b) result=(b, a)
```

Type check

1 + “2”

This actually works in MySQL

1 + 2

Query optimizer

What, Why, How

- What is query optimization?
- Why can we optimize a SQL query?
- How to do query optimization?

What is query optimization?

- Choose optimal execution plan for a given query
- Definition of “optimal”:
  - Fast speed(low latency): Intuitive
  - Cost effective: OLTP scenarios
  - High throughput: OLAP scenarios
  - The “Right” choice: Commercial slogan
  - etc.

Why can we optimize a SQL query?

- Equivalent alternatives based on relational algebra and algorithms
- A cost model to evaluate a query

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

$$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$$
$$\sigma_{\theta_1}(E_1 \bowtie_{\sigma_{\theta_2}} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

How to do query optimization?

- Build a framework to enumurate possible plans
- Write transformation rules
- Introduce a cost model to evaluate different plans
- Choose the most optimal one

The history of query optimization

- The first query optimizer in the world is IBM System R’s optimizer
- Background:
    - Disk is much slower than memory, seeking overhead is serious
    - Memory size is very small
    - Single CPU core(no parallelism)
- Cost model
    - Scan less tuples from disk as possible
    - Doesn’t count number of I/Os
    - Doesn’t measure CPU cost alongside I/O cost
- Dynamic programming based join reordering

The history of query optimization

- PostgreSQL is one of the most successful open source RDBMS in the world, with a long history(first released in 1996).

- Cost model:
    - *seq_page_cost*
    - *random_page_cost*
    - *cpu_tuple_cost*
    - *cpu_index_tuple_cost*
    - *cpu_operator_cost*

    - Total cost = Sum of corresponding number multiply the factor

- Dynamic programming based join reordering

- GQO(Genetic Query Optimizer): use genetic algorithm to reduce search space

The history of query optimization

- SQL Server, a commercial RDBMS developed by Microsoft and Sybase in 1990s.
- Goetz Graefe(the author of Volcano/Cascades) designed the Cascades query optimizer framework for SQL Server.
- This optimizer framework has been widely used in different query system developed by Microsoft(e.g. SQL Server, SQL Server PDW, Cosmos SCOPE, Synapse).
- The best query optimizer around the world(maybe)

Volcano/Cascades optimizer framework

- The Volcano Optimizer Generator: Extensibility and Efficient Search
- The Cascades Framework for Query Optimization
- An exploration framework to enumurate plans and evaluate the cost of them
- Concepts:
  - Logical operator: logical algebra, e.g. relational algebra in RDBMS
  - Physical operator: implementation of logical algebra, e.g. hash join, hash aggregation
  - Property: logical/physical property required/provided by operators, e.g. output columns
  - Transformation: rules to transform a logical operator into another equivalent logical operator
  - Implementation: rules to transform a logical algebra into physical operator
  - Enforcer: enforce operator to have specific physical property, e.g. adding a *Sort* operator to enforce *ordered* property

The Cascades principle

- Top-down exploring
- Pattern matching
- “Rule-based”
- Memoization

Open source cascades implementation

- Apache Calcite: a Volcano/Cascades style optimizer framework, widely used in Apache world(e.g. Drill, Flink)

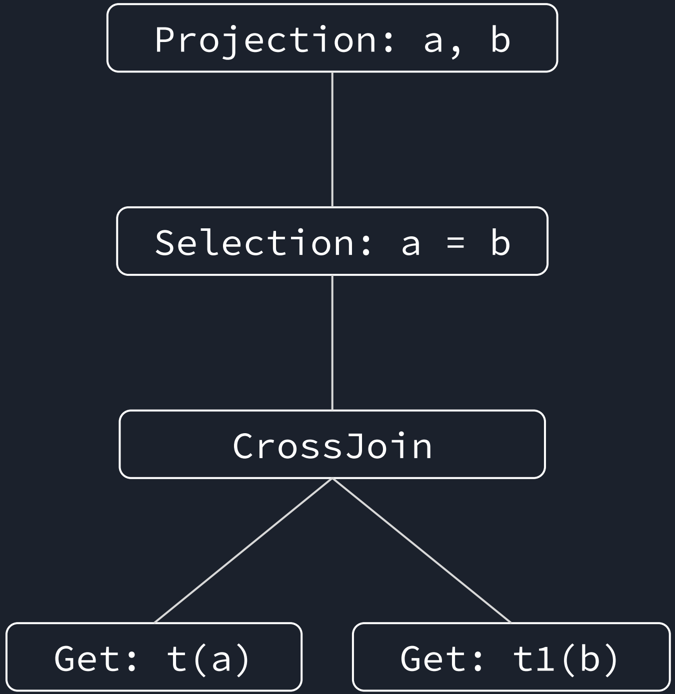
- GreenPlum Orca: optimizer component of GreenPlum, also used by HAWQ, Hologres, Alicloud ADB

- CockroachDB’s Cascades optimizer

- TiDB’s Cascades optimizer(not released yet)

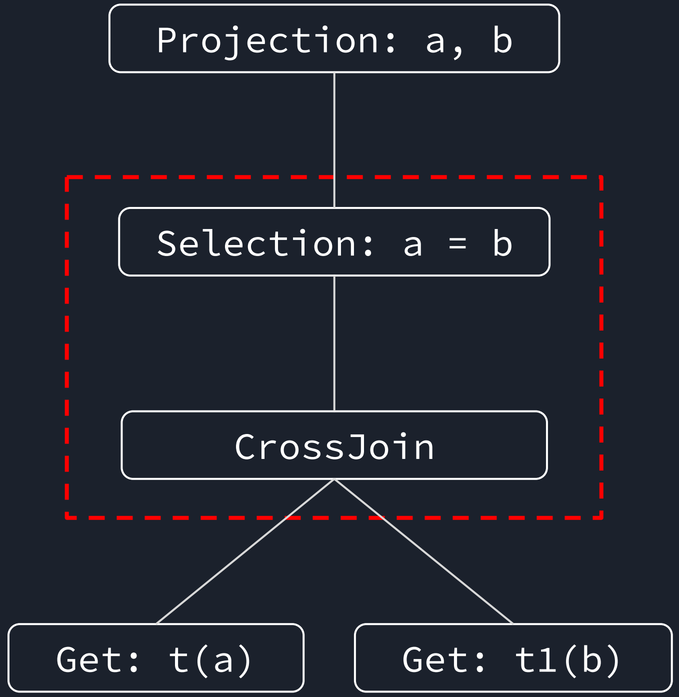
Cascading procedure

```
Group 1: Projection[2](a, b)

Group 2: Selection[3](a = b)

Group 3: CrossJoin[4,5]

Group 4: Get(t)

Group 5: Get(t1)
```



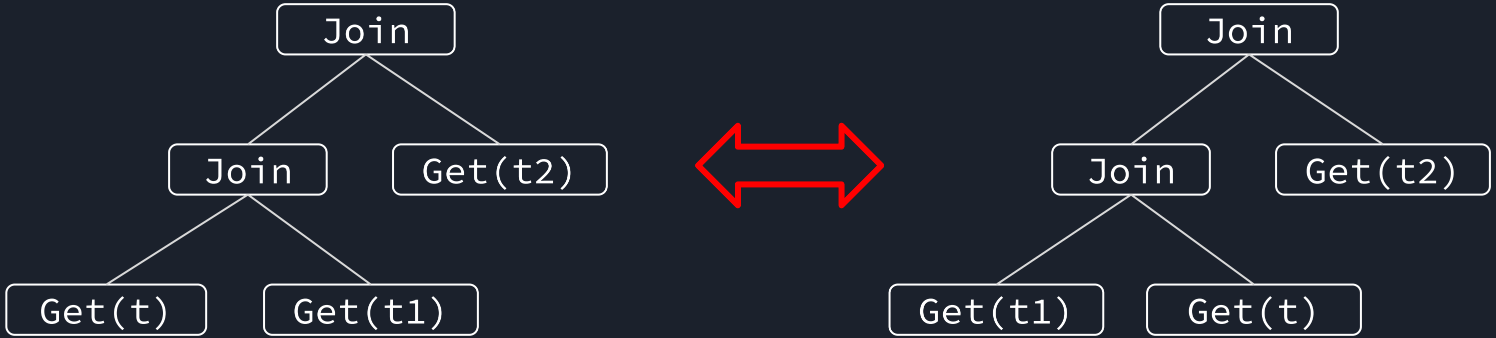
Predicate pushdown

```
Group 1: Projection[2](a, b)

Group 2: Selection[3](a = b)

         EquiJoin[4,5](a = b)

Group 3: CrossJoin[4,5]

Group 4: Get(t)

Group 5: Get(t1)
```



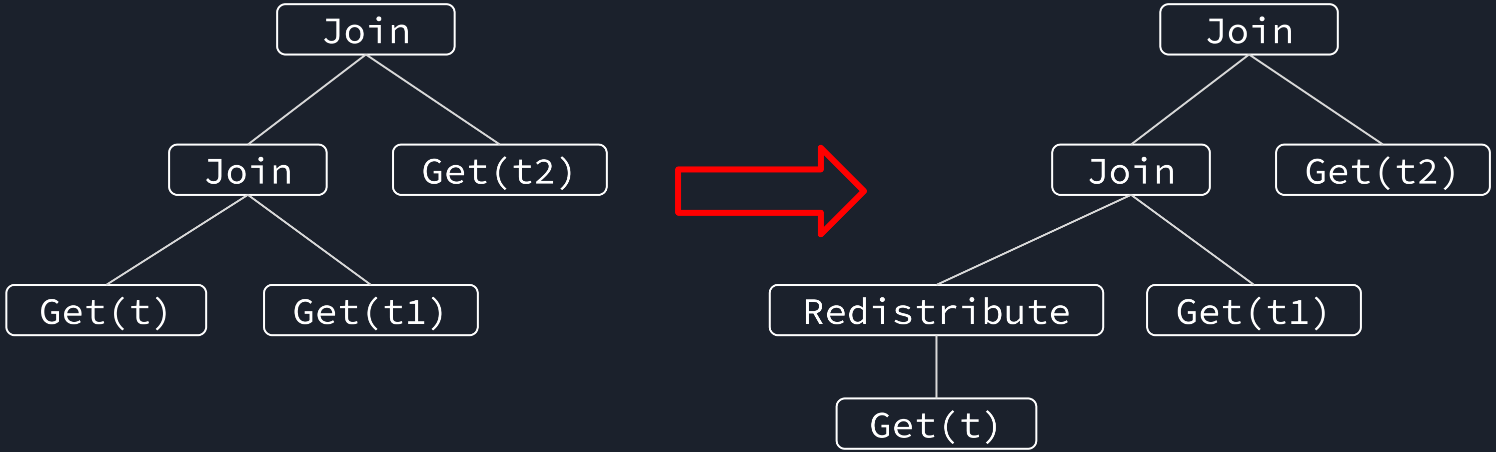
Join reordering

- Inner join operators are commutative
- Use Transformation rules to perform join reordering
- Paper: [Optimizing Join Enumeration in Transformation-based Query Optimizers] and GP Orca's paper



Distributed parallel execution

- Introduce partition physical property
- Use enforcer to add Exchange operator
- Paper: [Incorporating partitioning and parallel plans into the SCOPE optimizer]



Cost model

- Not a technique problem, but a philosophy problem

- For a MPP OLAP system, the factors can be catagorized as:
  - CPU cost: expression evaluation, join, aggregation, repartition
  - Disk I/O cost: fetching data from columnar storage
  - Network cost: data redistribution

- For a cloud data warehouse, there are more factors should be counted:
  - Cache hit ratio, since read data from object storage is much slower than from cache
  - Cost effective
  - etc.

Cost Estimation

- Predicate calculus and cost model is basis of cost estimation
- Statistics information is input of cost model
- Predicates:
    - Conjunction: AND
    - Disjunction: OR
    - Equivalence: =, <>
    - Nullability: IS NULL, IS NOT NULL
    - Range: IN, EXISTS, <, >
- Expression as predicate sometime, e.g. WHERE CAST(a AS BOOL)

Cardinality estimation

- Cardinality is the expectd output rows of an operator
- According to statistics information and predicate, we can calculate the cardinality of each operator in a plan tree
- Cardinality is important to compute cost

Selectivity

- Selectivity of a predicate indicate the ratio of output tuples to input tuples
- Suppose t has 100 rows, there is a predicate P, the result set has 30 rows
- Then the selectivity of P is 30%
- Lower is better

```
SELECT * FROM t WHERE P
```

A simple case of cardinality estimation

```
CREATE TABLE student (
    id INTEGER PRIMARY KEY,
    name VARCHAR(20) NOT NULL,
    age INTEGER NOT NULL,
    gender TINYINT(1) NOT NULL
)
```

```
SELECT * FROM student AS s
WHERE s.name = ‘Koji’
```

```
SELECT * FROM student AS s
WHERE s.age > 23
```

```
SELECT * FROM student AS s
WHERE s.id IN (11, 45, 14)
```

Equivalence

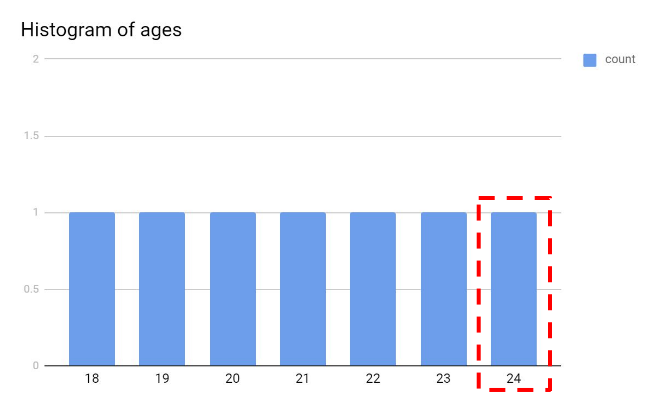
- P: s.age = 24

- Assume age is in range [18, 24], and the tuple number N is 7. Then there coulde be 7 kinds of values, and the cardinality in average is 1

- For a predicate A = Value, The selectivity Sel(A = Value) = C(R, A) / N

- In this case, Sel(age = 24) = C(student, age) / N = 1 / 7, selectivity is 1/7

```
SELECT * FROM student AS s
WHERE s.age = 24
```



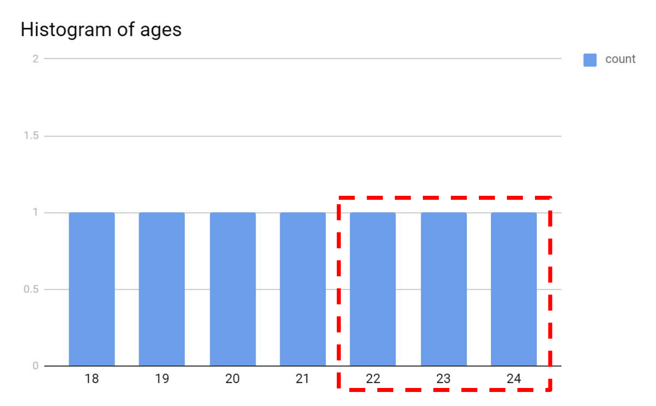
Range

- P: age > 21

- For a range predicate, the selectivity Sel(A > Value) = N(Max(A) - Value) / N(Max(A) - Min(A))

- Sel(age > 21) = 1 * (24 - 21) / 1 * (24 - 18) = 0.5

```
SELECT * FROM student AS s

WHERE s.age > 21
```



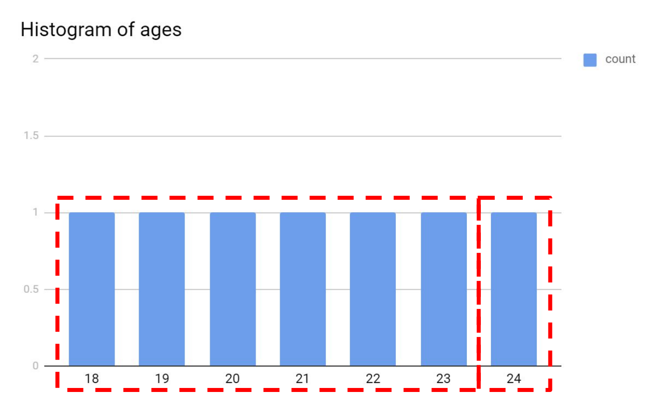
Negation

- P: age != 24

- For a negation predicate, the selectivity
Sel(A != Value) = 1 - Sel(A = Value)

- Sel(age != 21) = 1 - 1/7 = 6/7

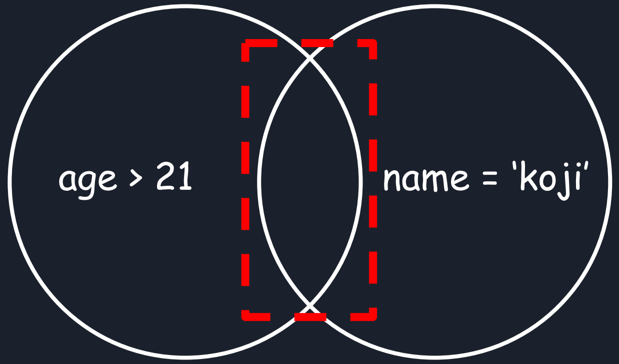
```
SELECT * FROM student AS s

WHERE s.age != 24
```



Conjunction

- P: age > 21 AND name = 'Koji'
- P1: age > 21
- P2: name = 'Koji'
- Sel(P) = Sel(P1) * Sel(P2)

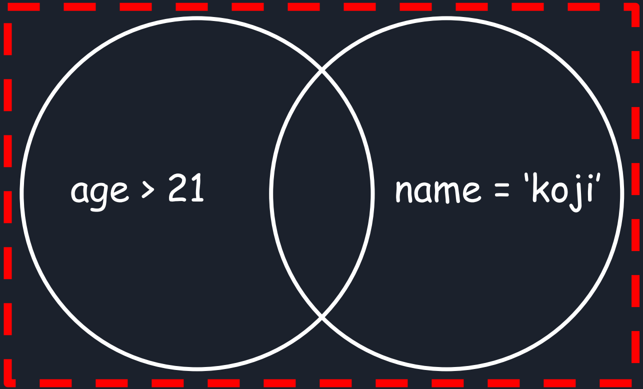
```
SELECT * FROM student AS s
WHERE s.age > 21
      AND s.name = 'Koji'
```



Disjunction

- P: age > 21 OR name = ‘Koji’
- P1: age > 21
- P2: name = ‘Koji’
- Sel(P) = Sel(P1) + Sel(P2) - Sel(P1) * Sel(P2)

```
SELECT * FROM student AS s
WHERE s.age > 21
     OR s.name = ‘Koji’
```



Predicate processing in practice

- Normalize predicate
- It’ s very tricky to normalize complicated predicates

```
a + 1 > 1
```

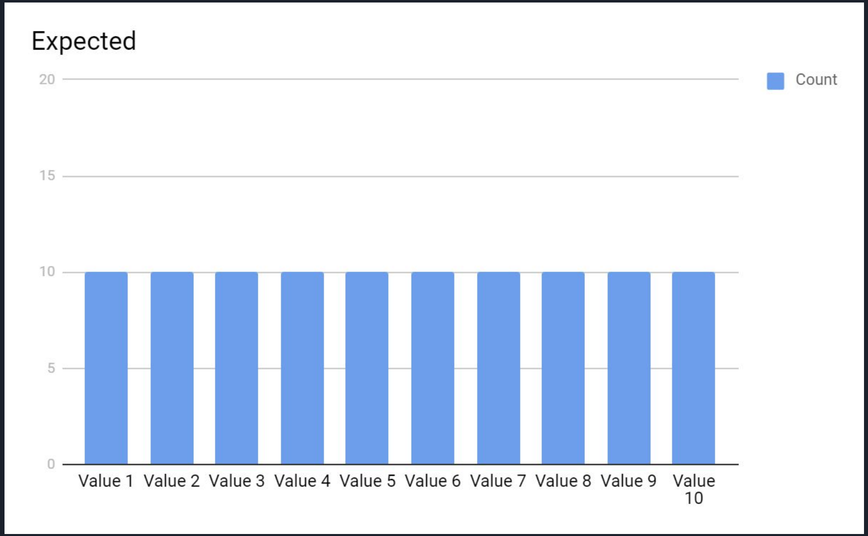
```
a > 0
```

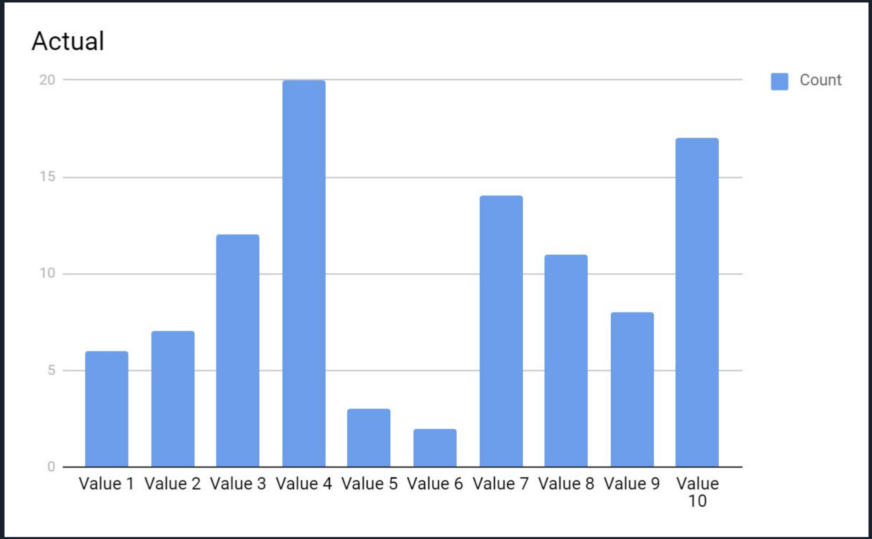
```
CAST(a AS UNSIGNED) < 1
```

```
a < 1 AND a > -1
```

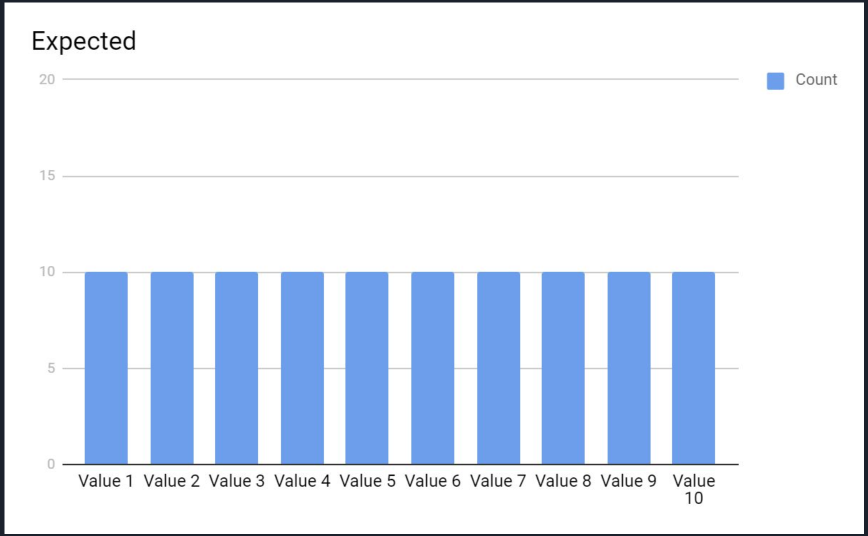
```
DATE_FORMAT(‘YYYY-MM-DD’, a) = ‘xxxxxxxxxx’
```

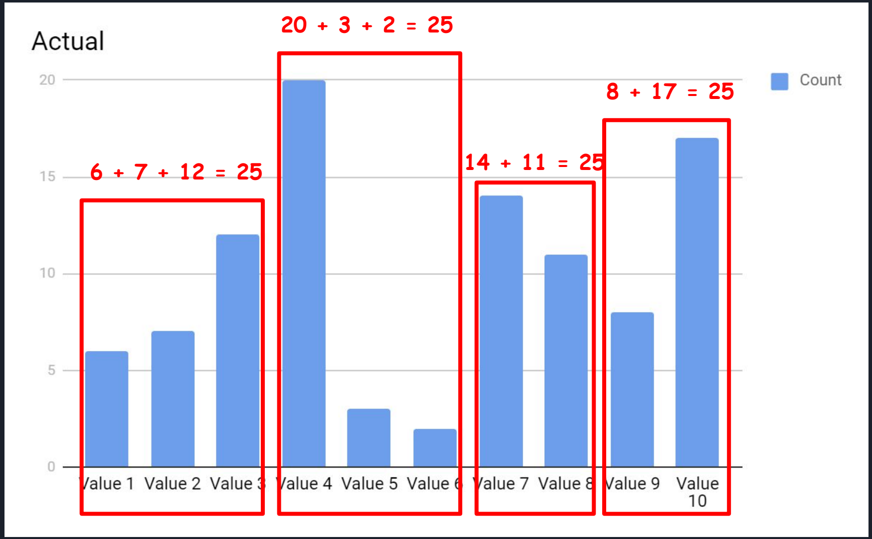
Ideal vs Reality



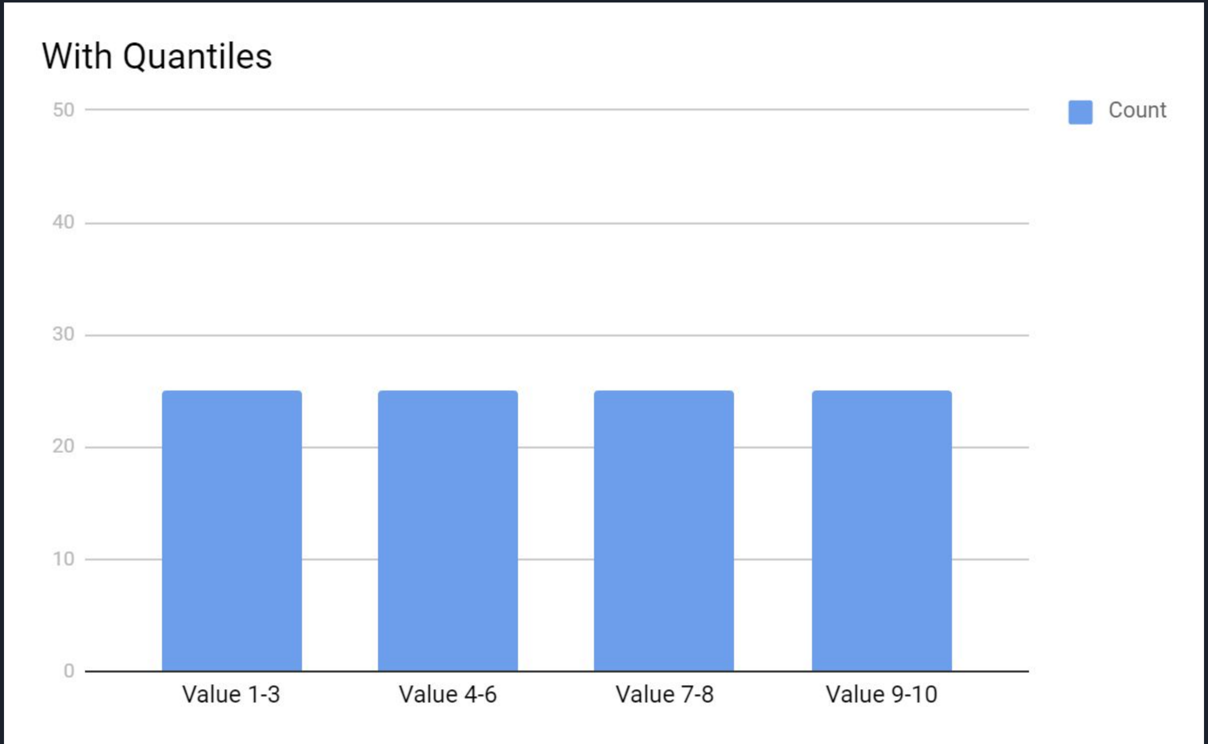


Equi-depth histogram





Equi-depth histogram



Histogram

- Distribution of data is always not uniform
- Equi-width V.S. Equi-depth

Data correlation

- Data are correlated
- brand = ‘Honda’ AND family = ‘Accord’
- gender = ‘female’ AND height > ‘180cm’
- This makes cardinality estimation much more complicated

What can we do with cardinality estimation?

- Join reordering by estimating the output rows of each Join operator
- Index selection, in OLTP scenarios
- Partition pruning, in OLAP scenarios

The challenge of query optimization in cloud data warehouse

- Cost estimation on very large data set
- Complicated cost model factors
- Different optimization approach, depends on design of storage system
- Testability, tracibility, debugability