UNB FREDERICTON

Big Data Systems (CS4545/CS6545)
Winter 2021

Query Processing

Suprio Ray

University of New Brunswick

Acknowledgement

Thanks to N. Koudas, Manos Papagelis, Jordan Halterman, Ramakrishna and Gehrke, Silberchatz, Korth and Sudarshan for materials in these slides

Outline

- Quick SQL *review*
- Query processing overview
- Relational Algebra *review*
- Introduction to Apache Calcite
- Join processing

Query processing language: SQL (Structured Query Language)

- Declarative
  - Say "what to do" rather than "how to do it"
    - Avoid data-manipulation details needed by procedural languages
  - Database engine figures out "best" way to execute query
    - Called "query optimization"
    - Crucial for performance: "best" can be a million times faster than "worst"
- Data independent
  - Decoupled from underlying data organization
    - Correctness always assured… performance not so much
  - SQL is standard and (nearly) identical among vendors

SQL Environment

- Catalog
  - A set of schemas that constitute the description of a database

```
                              List of databases
     Name      | Owner  | Encoding |   Collate   |    Ctype    | Access privileges
---------------+--------+----------+-------------+-------------+-------------------
 XDBSYS        | dbuser | UTF8     | en_CA.UTF-8 | en_CA.UTF-8 |
 __mytest__N1  | dbuser | UTF8     | en_CA.UTF-8 | en_CA.UTF-8 |
 __xtest__N1   | dbuser | UTF8     | en_CA.UTF-8 | en_CA.UTF-8 |
 dbgeo         | dbuser | UTF8     | en_CA.UTF-8 | en_CA.UTF-8 |
 postgres      | dbuser | UTF8     | en_CA.UTF-8 | en_CA.UTF-8 |
 template0     | dbuser | UTF8     | en_CA.UTF-8 | en_CA.UTF-8 | =c/dbuser         +
               |        |          |             |             | dbuser=CTc/dbuser
 template1     | dbuser | UTF8     | en_CA.UTF-8 | en_CA.UTF-8 | =c/dbuser         +
               |        |          |             |             | dbuser=CTc/dbuser
(7 rows)
```

SQL Environment

- Catalog
- Schema
  - The structure that contains descriptions of objects created by a user
  - A schema is a collection of related objects, including but not limited to base tables, views, constraints, domains, character sets, triggers and roles

```
                  List of relations
 Schema |           Name            |   Type   | Owner
--------+---------------------------+----------+--------
 public | arealm_merge_ca           | table    | dbuser
 public | arealm_merge_ca_gid_seq   | sequence | dbuser
 public | arealm_merge_ca_shall     | table    | dbuser
 public | arealm_merge_ca_shqtall   | table    | dbuser
 public | areawater_merge_ca        | table    | dbuser
 public | areawater_merge_ca_gid_seq | sequence | dbuser
 public | areawater_merge_ca_shall  | table    | dbuser
 public | areawater_merge_ca_shqtall | table    | dbuser
 public | edges_merge_ca            | table    | dbuser
 public | edges_merge_ca_gid_seq    | sequence | dbuser
 public | edges_merge_ca_shall      | table    | dbuser
 public | edges_merge_ca_shqtall    | table    | dbuser
 public | geography_columns         | view     | dbuser
 public | geometry_columns          | view     | dbuser
 public | raster_columns            | view     | dbuser
 public | raster_overviews          | view     | dbuser
 public | spatial_ref_sys           | table    | dbuser
(17 rows)
```

SQL Environment

- Catalog
- Schema
- Data Definition Language (DDL)
  - Commands that define a database, including creating, altering, and dropping tables and establishing constraints

```
dbgeo=# \d arealm_merge_ca
                              Table "public.arealm_merge_ca"
  Column  |          Type          |                   Modifiers
----------+------------------------+-----------------------------------------
 gid      | integer                | not null default nextval('arealm_me
 statefp  | character varying(2)   |
 countyfp | character varying(3)   |
 ansicode | character varying(8)   |
 areaid   | character varying(22)  |
 fullname | character varying(100) |
 mtfcc    | character varying(5)   |
 aland    | double precision       |
 awater   | double precision       |
 intptlat | character varying(11)  |
 intptlon | character varying(12)  |
 geom     | geometry(Polygon,4326) |
Indexes:
    "arealm_merge_ca_pkey" PRIMARY KEY, btree (gid)
    "arealm_merge_ca_geom_gist" gist (geom)
```

SQL Environment

- Catalog
- Schema
- Data Definition Language (DDL)
  - Commands that define a database, including creating, altering, and dropping tables and establishing constraints
- Data Manipulation Language (DML)
  - Commands that maintain and query a database
  - Commands for inserting, modifying and querying the data in the database

SQL Environment

- Catalog
- Schema
- Data Definition Language (DDL)
  - Commands that define a database, including creating, altering, and dropping tables and establishing constraints
- Data Manipulation Language (DML)
  - Commands that maintain and query a database
  - Commands for updating, inserting, modifying and querying the data in the database
- Data Control Language (DCL)
  - Commands that control a database, including administering privileges and committing data

Basic Single-Table Queries

- ```
  SELECT [DISTINCT] <column expression list>
    FROM <single table>
  [WHERE <predicate>]
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
    - Expression can be a column reference, or an arithmetic expression over column refs
  ```

Basic Single-Table Queries

- ```
  SELECT          S.name, S.gpa
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
    - Expression can be a column reference, or an arithmetic expression over column refs
  ```

ORDER BY

- ```
  SELECT  DISTINCT  S.name, S.gpa, S.age*2 AS a2
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
   ORDER BY S.gpa, S.name, a2;
  ```

- ORDER BY clause specifies that output should be sorted
  - Lexicographic ordering again!

- Obviously must refer to columns in the output
  - Note the AS clause for naming output columns!
  ```

ORDER BY

- ```
  SELECT  DISTINCT  S.name, S.gpa
    FROM Students S
   WHERE S.dept = ‘CS’
  [GROUP BY <column list>
   [HAVING <predicate>] ]
   ORDER BY S.gpa DESC, S.name ASC;
  ```

- Ascending order by default, but can be overridden
  - DESC flag for descending, ASC for ascending
  - Can mix and match, lexicographically
  ```

Aggregates

- ```
  SELECT [DISTINCT] AVERAGE(S.gpa)
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

- Before producing output, compute a summary (a.k.a. an *aggregate*) of some arithmetic expression

- Produces 1 row of output
  - with one column in this case

- Other aggregates: SUM, COUNT, MAX, MIN

- Note: can use DISTINCT inside the agg function
  - SELECT COUNT(DISTINCT S.name) FROM Students S
  - vs. SELECT DISTINCT COUNT (S.name) FROM Students S;
  ```

GROUP BY

```
SELECT [DISTINCT] AVERAGE(S.gpa), S.dept
  FROM Students S
[WHERE <predicate>]
 GROUP BY S.dept
 [HAVING <predicate>]
[ORDER BY <column list>]
```

- Partition the table into groups that have the same value on GROUP BY columns a.k.a. *grouping key*
  - Can group by a list of columns
- Produce an aggregate result per group
  - Cardinality of output = # of distinct group values
- Note: can put grouping keys in SELECT list
  - For aggregate queries, SELECT list can contain aggs and grouping keys only!

HAVING

- ```
  SELECT [DISTINCT] AVERAGE(S.gpa), S.dept
    FROM Students S
  [WHERE <predicate>]
   GROUP BY S.dept
    HAVING COUNT(*) > 25
  [ORDER BY <column list>]
  ```

- The HAVING predicate is applied *after* grouping and aggregation
  - Hence can contain anything that could go in the SELECT list
  - i.e. aggs or GROUP BY columns (i.e. grouping keys)

- HAVING can only be used in aggregate queries

- It's an optional clause
  ```

Putting it all together

```
SELECT S.dept, COUNT(*)
  FROM Students S
 WHERE S.gender = "F"
 GROUP BY S.dept
  HAVING COUNT(*) > 25
 ORDER BY S.dept;
```

Multi-relation Queries

- Interesting queries often combine data from more than one relation.
- We can address several relations in one query by listing them all in the FROM clause.
- Distinguish attributes of the same name by “<relation>.<attribute>” .

Types of multi-relation queries

- **Join**–a relational operation that causes two or more tables with a common domain to be combined into a single table or view

Types of multi-relation Queries

- Join

- **Inner-join**–a join that will only return rows from each table that have matching rows in the other
  - For each customer who placed an order, what is the customer's name and order number?

```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
   CustomerName, OrderID
FROM Customer_T INNER JOIN Order_T ON
   Customer_T.CustomerID = Order_T.CustomerID
ORDER BY OrderID;
```

```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
   CustomerName, OrderID
      FROM Customer_T, Order_T
         WHERE Customer_T.CustomerID = Order_T. CustomerID
         ORDER BY OrderID
```

Types of multi-relation Queries

- Join
- Inner-join
- **Outer join**–a join in which rows that do not have matching values in common columns are nonetheless included in the result table
  - List the customer name, ID number, and order number for all customers. Include customer information <u>even for customers that do not have an order</u>?

```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
  CustomerName, OrderID
    FROM Customer_T, Order_T
      WHERE Customer_T.CustomerID = Order_T. CustomerID
      ORDER BY OrderID
```

Types of multi-relation Queries

- Join
- Inner-join
- **Outer join**–a join in which rows that do not have matching values in common columns are nonetheless included in the result table
  - List the customer name, ID number, and order number for all customers. Include customer information <u>even for customers that do not have an order</u>?

```
SELECT Customer_T.CustomerID, CustomerName, OrderID
   FROM Customer_T LEFT OUTER JOIN Order_T
   WHERE Customer_T.CustomerID = Order_T. CustomerID;
```

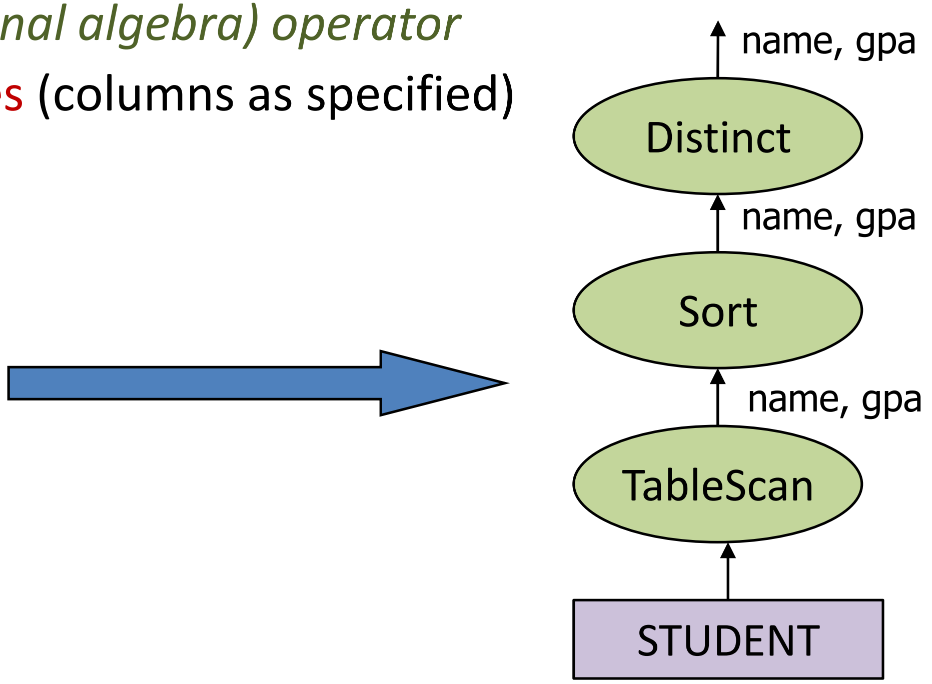
Outline

- Quick SQL review
- Query processing overview
- Relational Algebra review
- Introduction to Apache Calcite
- Join processing

Query Processing Overview

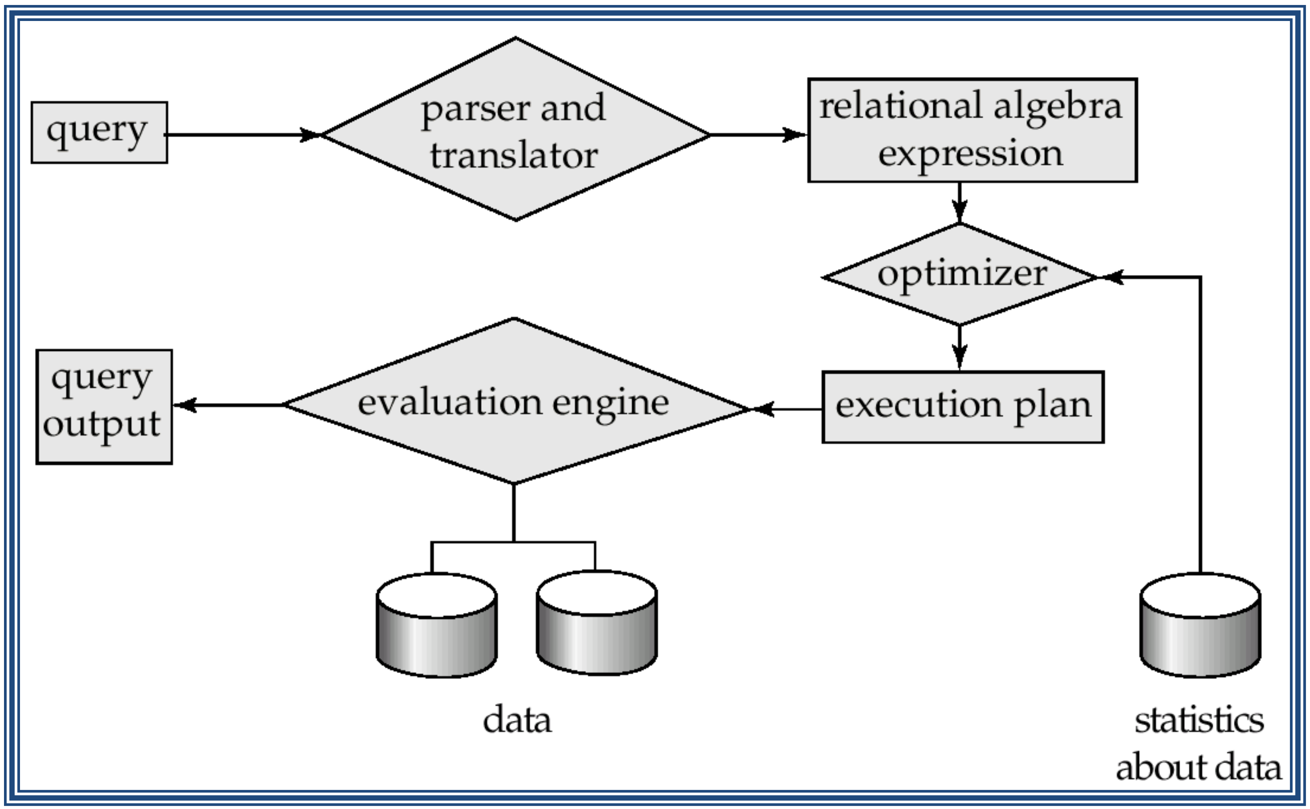
- Query processing requires translating SQL to a special internal language
  - Query Plans
- The *query executor* is an *interpreter* for query plans
- Think of query plans as "box-and-arrow" *dataflow* diagrams
  - Each box implements a (*relational algebra) operator*
  - Edges represent a flow of tuples (columns as specified)

```
SELECT DISTINCT name, gpa
  FROM Students
```



Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - Translate the query into its internal form. This is then translated into relational algebra expressions (query plans)
  - Parser checks syntax, verifies relations

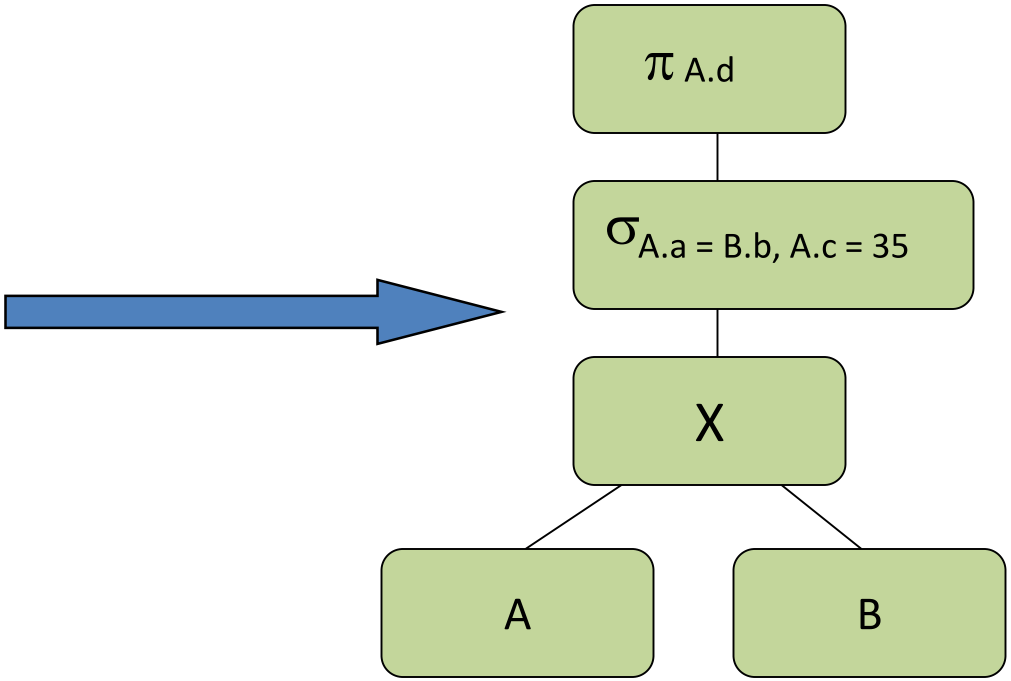
- Query optimization
  - The query optimizer chooses the best query plan (execution plan)

- Execution
  - The query-execution engine takes a execution plan, executes that plan, and returns the answers to the query.

Query plan

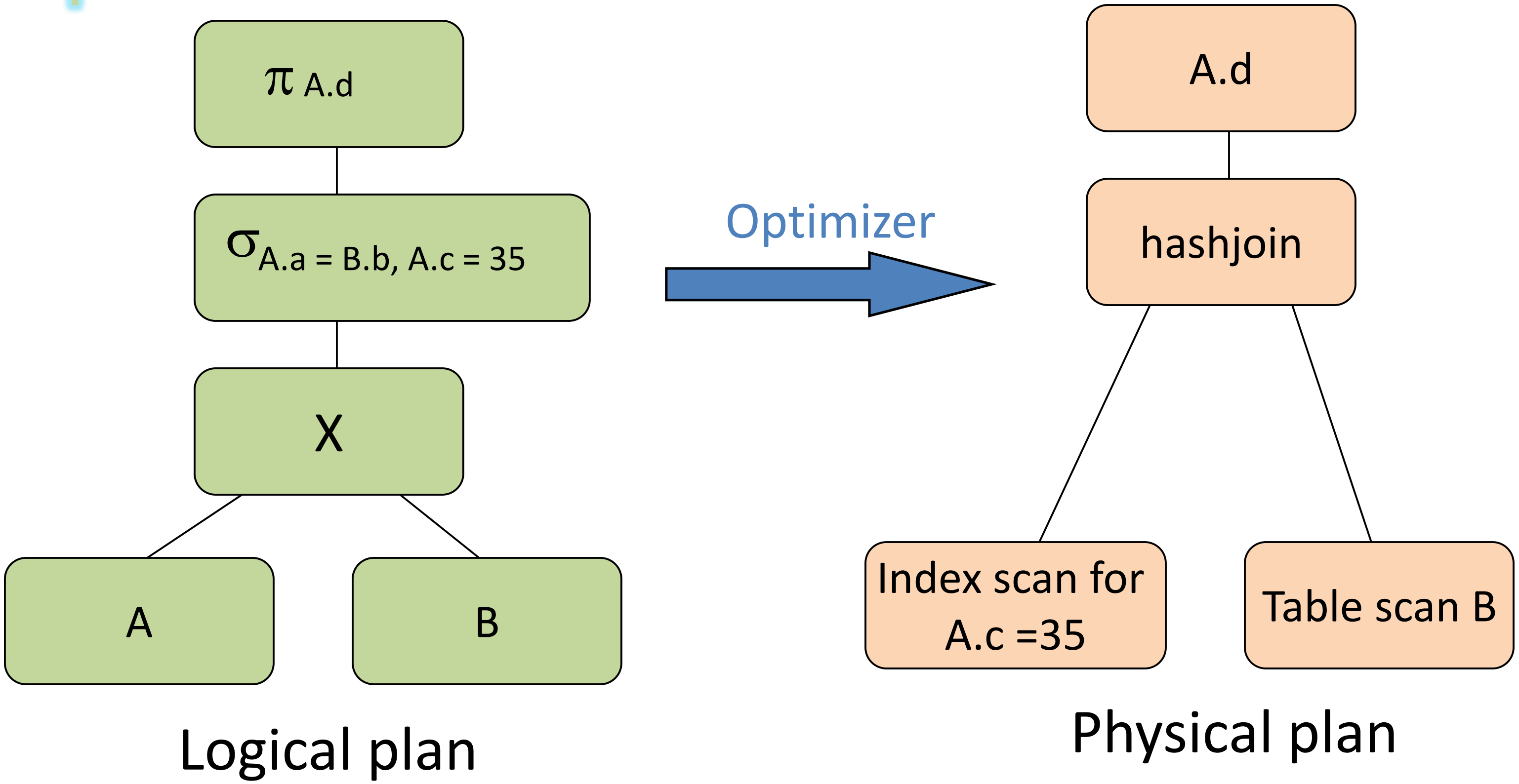
TABLES:
A(a,c,d)
B(b,e,f)

QUERY:
SELECT A.d
FROM A, B
WHERE A.a = B.b
AND A.c = 35



- Relational algebra for SQL very well understood

Query Optimization



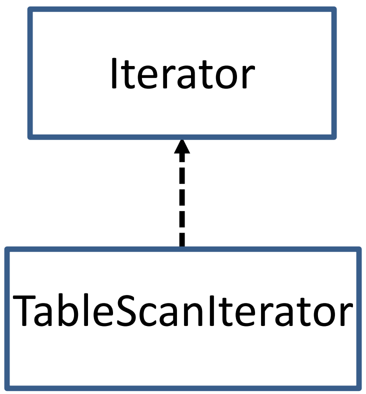
- logical, e.g., push down cheap predicates
- enumerate alternative plans, apply cost model
- use search heuristics to find cheapest plan

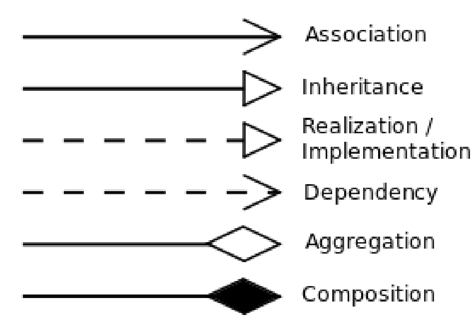
Iterators

- The **relational operators** can be <u>implementation</u> of the interface Iterator:

```
interface Iterator {
    void init();
    tuple next();
    void close();
   // additional states go here
   …   …   …
}

class TableScanIterator implements
Iterator {
    void init() {…}
    tuple next()  {…}
    void close()  {…}
   // additional states go here
    …   …   …
}
```





UML relations notation

Outline

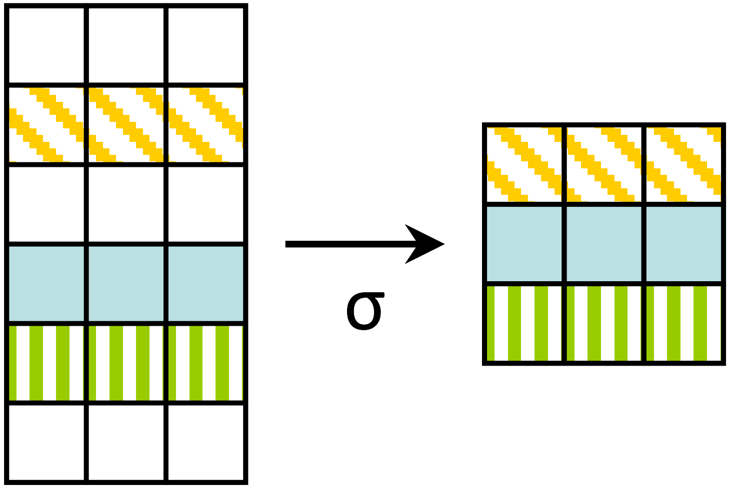
- Quick SQL review
- Query processing overview
- Relational Algebra review
- Introduction to Apache Calcite
- Join processing

Relational Algebra summary

- Selection: $\sigma_p(R)$
  - Returns all rows in R that satisfy p
- Projection: $\pi_C(R)$
  - Returns all rows in R projected to columns in C
    - In strict relational model, remove duplicate rows
    - In SQL, preserve duplicates (multiset/bag semantics)
- Cartesian Product: $R \times S$
- Union: $R \cup S$ Intersection: $R \cap S$
  - Note: R, S must have matching schema
- Join: $R \bowtie_p S = \sigma_p(R \times S)$

Unary operators: select (σ)

- $\sigma_P(R)$ outputs tuples of R which satisfy P
- same schema as R



Removes unwanted rows from relation

Unary operators: select (σ) example

Employees

Surname	FirstName	Age	Salary
Smith	Mary	25	2000
Black	Lucy	40	3000
Verdi	Nico	36	4500
Smith	Mark	40	3900

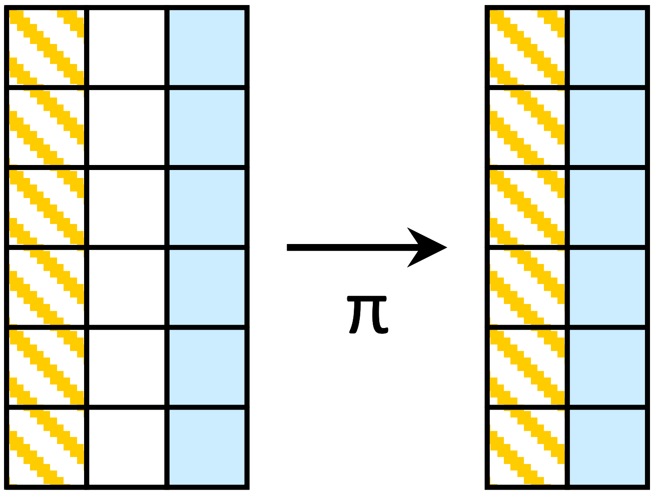
Q. Show the employees with salary greater than 4000 and age less than 30

$\sigma_{Age<30 \vee Salary>4000}$ **(Employees)**

Surname	FirstName	Age	Salary
Smith	Mary	25	2000
Verdi	Nico	36	4500

Unary operators: project (π)

- $\pi_Y(R)$ outputs a subset Y of the set of attributes X of relation R



Removes unwanted columns from relation

Unary operators: project (π) example

Employees

Surname	FirstName	Department	Head
Smith	Mary	Sales	De Rossi
Black	Lucy	Sales	De Rossi
Verdi	Mary	Personnel	Fox
Smith	Mark	Personnel	Fox

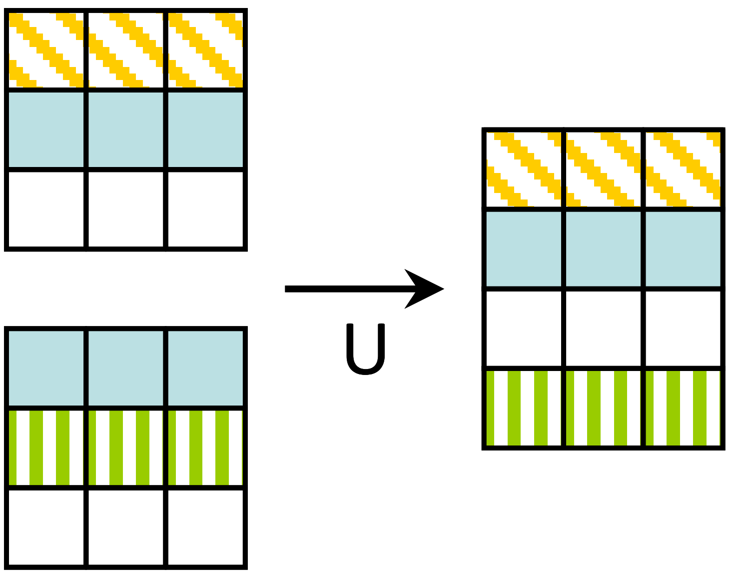
Q. Show the surname and firstname of the employees

$\pi_{\text{Surname, FirstName}}(\text{Employees})$

Surname	FirstName
Smith	Mary
Black	Lucy
Verdi	Mary
Smith	Mark

Additive operators (∪, ∩, -)

- Standard set operators
- Operate on tuples within input relations, but not on schema



Additive operators: Union (U)

Graduates

Number	Surname	Age
7274	Robinson	37
7432	O'Malley	39
9824	Darkes	38

Managers

Number	Surname	Age
9297	O'Malley	56
7432	O'Malley	39
9824	Darkes	38

Graduates ∪ Managers

Number	Surname	Age
7274	Robinson	37
7432	O'Malley	39
9824	Darkes	38
9297	O'Malley	56

Additive operators: Intersection (∩)

Graduates

Number	Surname	Age
7274	Robinson	37
7432	O'Malley	39
9824	Darkes	38

Managers

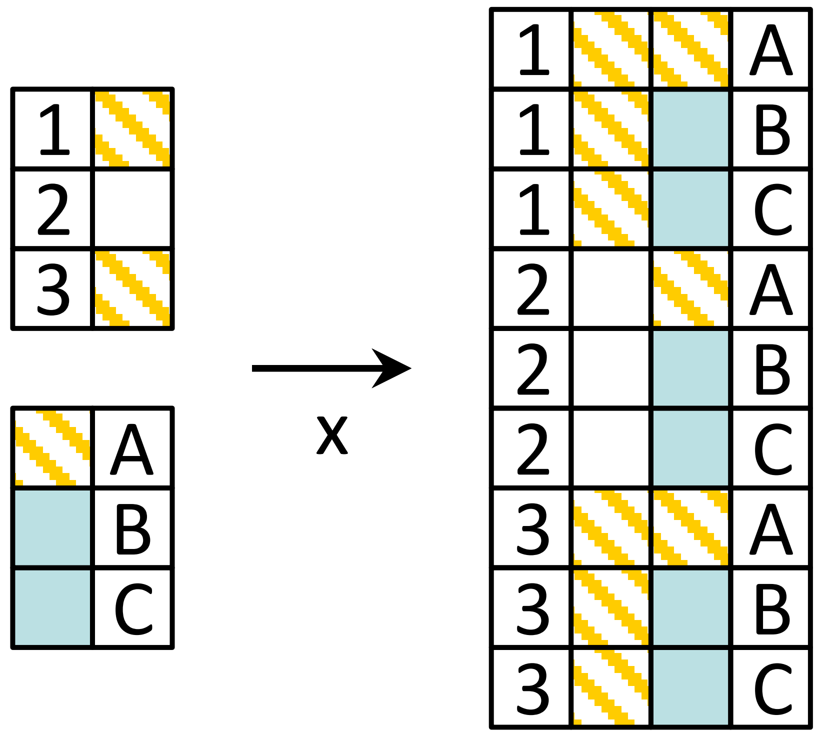
Number	Surname	Age
9297	O'Malley	56
7432	O'Malley	39
9824	Darkes	38

Graduates ∩ Managers

Number	Surname	Age
7432	O'Malley	39
9824	Darkes	38

Cartesian product (×)

- The outcome of combining every record in R with every record in S
- T = R × S contains every pairwise combination of R and S tuples
  - schema(T) = schema(R) U schema(S)



Cartesian product (×) example

Employees

Employee	Project
Smith	A
Black	A
Black	B

Projects

Code	Name
A	Venus
B	Mars

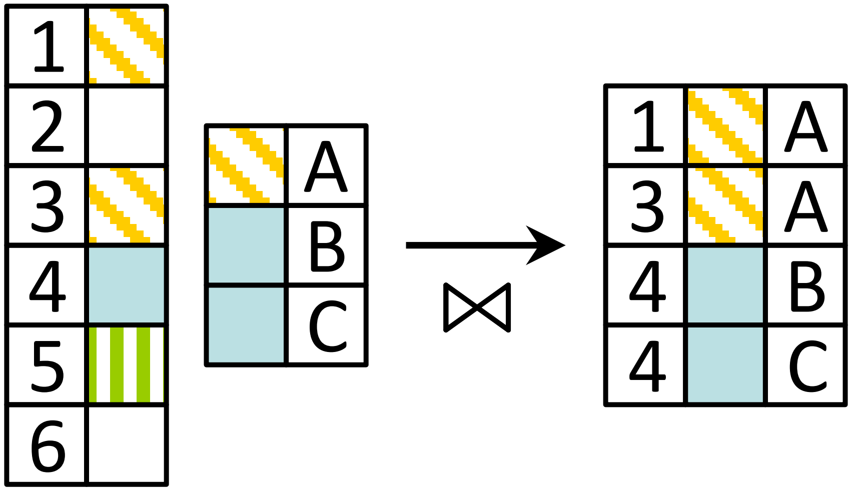
Employees × **Projects**

Employee	Project	Code	Name
Smith	A	A	Venus
Black	A	A	Venus
Black	B	A	Venus
Smith	A	B	Mars
Black	A	B	Mars
Black	B	B	Mars

Natural join (⋈)

- T = R ⋈ S merges tuples from R and S having equal values where their schemas overlap (join attributes)
  - T Schema: Union of schemas
    schema(R) ∩ schema(S) ≠ Ø

- Special cases
  - No schema overlap: ×
  - Full schema overlap: ∩



Natural join (⋈) example

r_1

Employee	Department
Smith	sales
Black	production
White	production

r_2

Department	Head
production	Mori
sales	Brown

$r_1 \bowtie r_2$

Employee	Department	Head
Smith	sales	Brown
Black	production	Mori
White	production	Mori

Theta join

- Written as T = R $\bowtie_{\theta}$ S
  - Outputs pairwise combinations of tuples which satisfy θ
- Most general join
  - Arbitrary join predicate (not just equality)

Theta join example

Car

Car	CarPrice
CarA	20000
CarB	30000
CarC	50000

Boat

Boat	BoatPrice
BoatA	10000
BoatB	40000
BoatC	60000

Q. select the cars and boats where car price is higher than boat price

$\text{Car} \bowtie_{\text{CarPrice}>\text{BoatPrice}} \text{Boat}$

Car	CarPrice	Boat	BoatPrice
CarA	20000	BoatA	10000
CarB	30000	BoatA	10000
CarC	50000	BoatA	10000
CarC	50000	BoatB	40000

Equijoin

- Special case of theta join
- Written as $R \bowtie_{A=X,B=Y,\ldots} S$
  - Attribute names in R and S can differ
  - Still compare values for equality
- Like natural join, but using arbitrary attributes
  - Very common due to *foreign keys* in relations

Equijoin example

Employees

Employee	Project
Smith	A
Black	A
Black	B

Projects

Code	Name
A	Venus
B	Mars

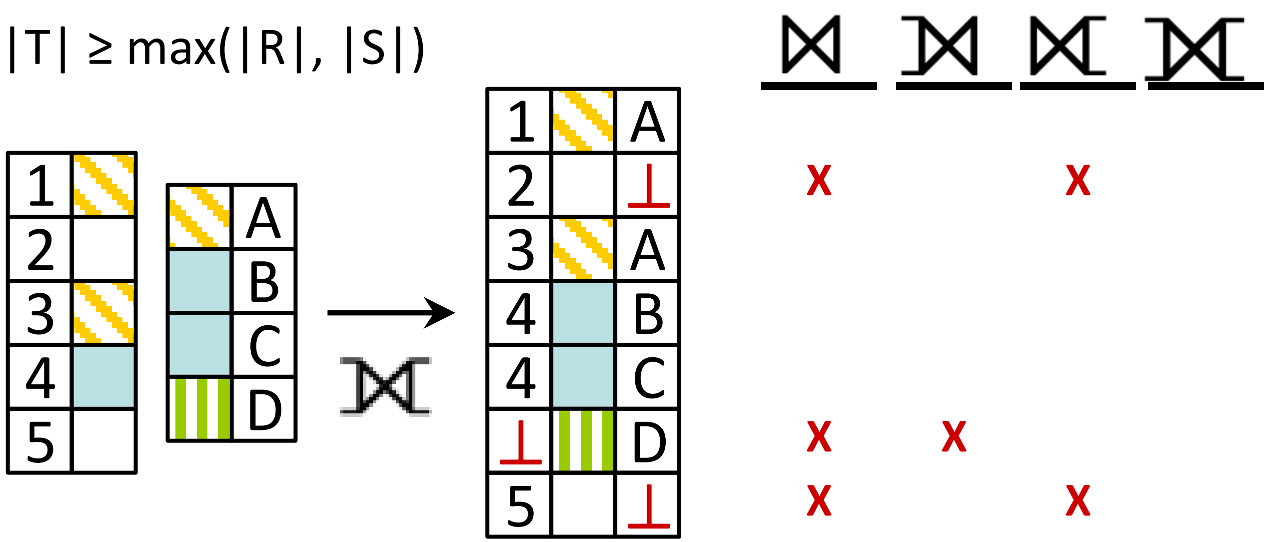
Q. select employees and the projects they work on

Employees $\bowtie_{Project=Code}$ Projects

Employee	Project	Code	Name
Smith	A	A	Venus
Black	A	A	Venus
Black	B	B	Mars

Outer join (⟗)

- T = R ⟗ S computes the “outer” join of R (left) and S (right)
  - Like normal join, but all tuples from R and S appear in output
  - Pad (left, right, or all) dangling tuples with ⊥ or NULL
    - **LEFT** — Tuples in inner join padded with tuples in R that have no matching tuples in S.
    - **RIGHT** — Tuples in inner join padded with tuples in S that have no matching tuples in R.
    - **FULL** — Tuples in inner join padded with tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R.
  - *Natural, equi-, and theta- variants still apply*
  - |T| ≥ max(|R|, |S|)



Outer join (⟕) examples

r_1

Employee	Department
Smith	sales
Black	production
White	production

r_2

Department	Head
production	Mori
purchasing	Brown

r_1 ⟕ r_2

Employee	Department	Head
Smith	sales	NULL
Black	production	Mori
White	production	Mori

r_1 ⟖ r_2

Employee	Department	Head
Black	production	Mori
White	production	Mori
NULL	purchasing	Brown

r_1 ⟗ r_2

Employee	Department	Head
Smith	Sales	NULL
Black	production	Mori
White	production	Mori
NULL	purchasing	Brown

Sorting (τ)

- τ_L(R) sorts tuples in R on list of attributes L
  - If L is A1, A2, …,An tuples sorted first by A1. Ties are broken based on A2;…; Ties that remain after An broken arbitrarily.
  - Default: ascending order; With '-' in front: descending order
- Example: $\tau_{\text{-Count, Make}}$ (R)

Make	Count
Toyota	2
Honda	3
Ford	3

$\xrightarrow{\tau}$

Make	Count
Ford	3
Honda	3
Toyota	2

Descending count

Alphabetical order when count is equal

Grouping (Γ)

- Aggregate functions
  - min, max, sum, count, average, …

- $\Gamma_{A,B,C,f(X),g(Y),h(Z)}(R)$ computes aggregate values using some attributes as a grouping key
  - Implicit projection (drops unreferenced attributes)
  - A, B, C is the *grouping key*
  - X, Y, Z are *attributes* to aggregate
  - *f*, *g*, *h* are *aggregating* functions to apply

Grouping Example

- Answer the following query
  - "List employees and their total sales in descending order"
  - $\tau_{\text{-Total}}(\rho_{\text{Name,Total}}(\Gamma_{\text{Name,sum(Value)}}(\text{Emp} \bowtie \text{Sales})))$

Emp

EID	Name
1	Mary
2	Xiao
3	Jaspreet

$\bowtie$

Sales

EID	Value	…
1	20	…
3	10	…
3	15	…

=

Name	Total
Jaspreet	25
Mary	20

Schema for Additional Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)

Boats (*bid*: integer, *bname*: string, *color*: string)

Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

Src: Tulane Univ.

Example 1

Find names of sailors who’ve reserved boat #103

Example 2

Find sailors who've reserved a red or a green boat

Exercise

Find sailors who’ve reserved a red <u>and</u> a green boat

Outline

- Quick SQL review
- Query processing overview
- Relational Algebra review
- Introduction to Apache Calcite
  (Some slides are from J. Halterman)
- Join processing

What is Apache Calcite?

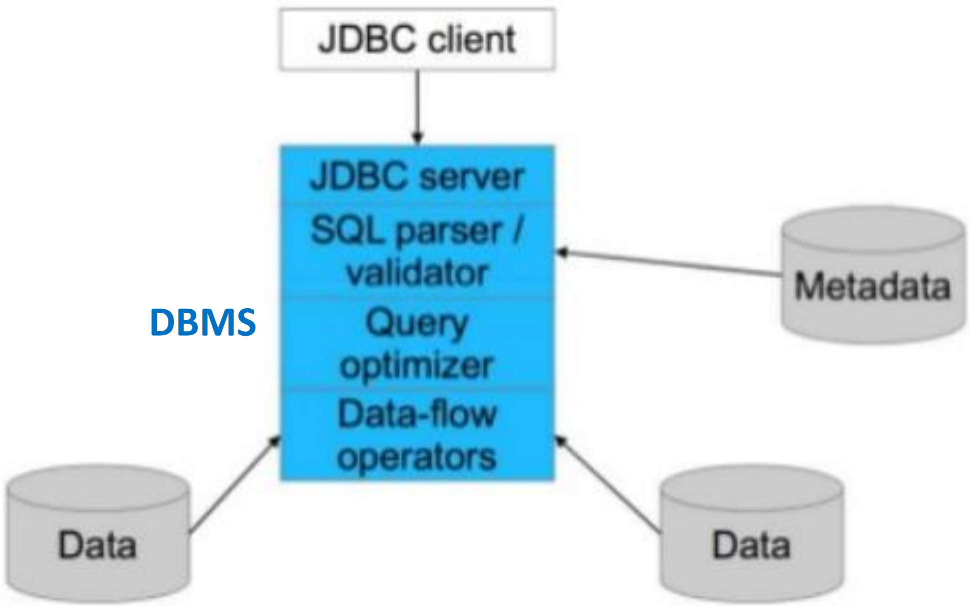
- A framework for building SQL databases
- Developed over more than ten years
- Written in Java
- Previously known as Optiq
- Previously known as Farrago
- Became an Apache project in 2013
- Led by Julian Hyde at Hortonworks

Projects using Calcite

- Apache Hive
- Apache Drill
- Apache Flink
- Apache Phoenix
- Apache Samza
- Apache Storm
- Apache everything…

What is Apache Calcite?

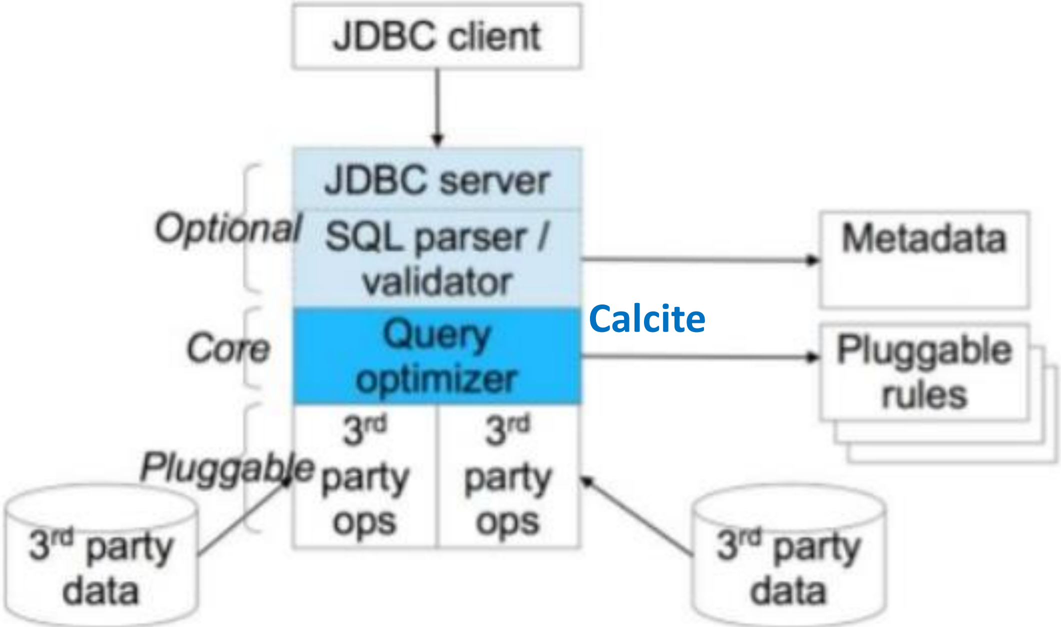
Conventional DBMS architecture



Src: J. Hyde

What is Apache Calcite?

Apache Calcite architecture



Src: J. Hyde

Stages of query execution

01

Parse

Queries are parsed using a JavaCC generated parser

02

Validate

Queries are validated against known database metadata

03

Optimize

Logical plans are optimized and converted into physical expressions

04

Execute

Physical plans are converted into application-specific executions

Outline

- Quick SQL review
- Query processing overview
- Relational Algebra review
- Introduction to Apache Calcite
    - Components
- Join processing

Components of Calcite

- **Catalog** - Defines metadata and namespaces that can be accessed in SQL queries
- **SQL parser** - Parses valid SQL queries into an abstract syntax tree (AST)
- **SQL validator** - Validates abstract syntax trees against metadata provided by the catalog
- **Query optimizer** - Converts AST into logical plans, optimizes logical plans, and converts logical expressions into physical plans

Outline

- Quick SQL review
- Query processing overview
- Relational Algebra review
- Introduction to Apache Calcite
    - Components
    - Query plan and optimization
- Join processing

Query Plans

- Query plans represent the steps necessary to execute a query

Query Plans

- Query plans represent the steps necessary to execute a query

```sql
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

Query Plans

- Query plans represent the steps necessary to execute a query

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

table scan

table scan

Query Plans

- Query plans represent the steps necessary to execute a query

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

table scan

inner join

table scan

Query Plans

- Query plans represent the steps necessary to execute a query

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

table scan

filter

inner join

table scan

Query Plans

- Query plans represent the steps necessary to execute a query

project

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

table scan

inner join

table scan

filter

Query Plans

- Query plans represent the steps necessary to execute a query

project

```sql
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

table scan

filter

inner join

table scan

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

Query Optimization

- Optimize logical plan
- Goal is typically to try to reduce the amount of data that must be processed early in the plan
- Convert logical plan into a physical plan
- Physical plan is engine specific and represents the physical execution stages

Query Optimization

- Prune unused fields
- Merge projections
- Convert subqueries to joins
- Reorder joins
- Push down projections
- Push down filters

Query Optimization

```sql
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

Query Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

Query Optimization

```sql
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

Query Optimization

```sql
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

push down project

Query Optimization

```
SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
  FROM users u INNER JOIN orders o ON u.id = o.user_id
  WHERE u.id > 50
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalFilter(condition=[>($0, 50)])
    LogicalJoin(condition=[=($0, $6)], joinType=[inner])
      LogicalTableScan(table=[[USERS]])
      LogicalTableScan(table=[[ORDERS]])
```

```
LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
  LogicalJoin(condition=[=($0, $6)], joinType=[inner])
    LogicalProject(ID=[$0], NAME=[$1])
      LogicalFilter(condition=[>($0, 50)])
        LogicalTableScan(table=[[USERS]])
    LogicalProject(ID=[$0], USER_ID=[$1])
      LogicalTableScan(table=[[ORDERS]])
```

push down project

push down filter

Outline

- Quick SQL review
- Query processing overview
- Relational Algebra review
- Introduction to Apache Calcite
    - Components
    - Query plan and optimization
    - Key concepts
- Join processing

Key Concepts

- Relational algebra - `RelNode`
  - A relational algebra expression
- Row expressions - `RexNode`
  - A row-level expression (e.g. Projection fields, Filter condition)
- Traits - `RelTrait`
  - A trait of a relational expression that does not alter execution
- Conventions –
  - used to represent a single data source (`SparkConvention`, `JdbcConvention` etc)
- Rules - `RelOptRule`
  - Rules are used to modify query plans
- Planners - `RelOptPlanner`
  - Represents the query planner

Key Concepts

Relational algebra	`RelNode`
Row expressions	`RexNode`
Traits	`RelTrait`
Conventions	`Convention`
Rules	`RelOptRule`
Planners	`RelOptPlanner`

Relational Algebra

- `RelNode` represents a relational expression
- Largely equivalent to Spark's `DataFrame` methods
- Logical algebra
- Physical algebra

Outline

- Quick SQL review
- Query processing overview
- Relational Algebra review
- Introduction to Apache Calcite
  - Components
  - Query plan and optimization
  - Key concepts
  - Relational algebra builder
- Join processing

Relational Algebra APIs

Operation	**RA Operator**	**Relbuilder method**	**Method argument**
Table scan		scan	
Project	$\pi_C(R)$	project	
Select	$\sigma_p(R)$	filter	
Join (inner) Join(left out.) Join (right out.) Join (full out.)	⋈ ⟕ ⟖ ⟗	join	JoinRelType.INNER JoinRelType.LEFT JoinRelType.RIGHT JoinRelType.FULL
Union	$\cup$	union	
Intersection	$\cap$	intersect	
Grouping	Γ	aggregate	
Sorting	τ	sort	
Rename	ρ	as	
And	$\wedge$	and	
Or	$\vee$	or	

Steps to using building and running Relational Algebra expressions

```
import org.apache.calcite.tools.RelBuilder;
import org.apache.calcite.tools.RelRunners;

// Create a builder. The config contains a schema mapped
final FrameworkConfig config = buildConfig(); // see code
final RelBuilder builder = RelBuilder.create(config);

// Build RA expression for query: select * from COURSE
builder.scan("COURSE");

// Returns the final relational algebra expression
final RelNode node = builder.build();

// execute the query plan
try  {
    final PreparedStatement preparedStatement =
                                    RelRunners.run(node, calConn);
    ResultSet rs =  preparedStatement.executeQuery();
    while (rs.next()) { // do something }
    rs.close();
} catch (SQLException e) {
}
```

Examples of building RA expression: example 1

Schema

```
COURSE(COURSEID:int, TITLE:string, CATEGORYID:int)
CCATEGORY(CATID:int, CATNAME:string
```

Query

```
-- Show the title of the course where courseid = 2
select TITLE from COURSE where COURSEID = 2

// Build RA expression for the above query
builder
.scan("COURSE")
.filter(  builder.equals(builder.field("COURSEID"), builder.literal(2))  )
// or
//.filter(  builder.call(SqlStdOperatorTable.EQUALS,
builder.field("COURSEID"), builder.literal(2) ) )
.project(builder.field("TITLE"));


// See Javadoc for RelBuilder APIs
// https://calcite.apache.org/apidocs/org/apache/calcite/tools/RelBuilder.html
```

Examples of building RA expression: example 2

Schema

```
COURSE(COURSEID:int, TITLE:string, CATEGORYID:int)
CCATEGORY(CATID:int, CATNAME:string
```

Query

```
-- Show the coursed and title of the first 5 courses sorted by the
course
```

```
// Build RA expression for the above query
builder
.scan("COURSE")
.sort(  builder.field("COURSEID")  )
.limit(0, 5) // offset 0, limit 5
.project(builder.field("COURSEID"), builder.field("TITLE"));

// See Javadoc for RelBuilder APIs
// https://calcite.apache.org/apidocs/org/apache/calcite/tools/RelBuilder.html
```

Examples of building RA expression: example 3

Schema

```
COURSE(COURSEID:int, TITLE:string, CATEGORYID:int)
CCATEGORY(CATID:int, CATNAME:string
```

Query

```
-- Show the number of courses in each course category
SELECT CATEGORYID, count(*) AS C,
FROM COURSE
GROUP BY CATID
```

```
// Build RA expression for the above query
builder
.scan("COURSE")
.aggregate(builder.groupKey("CATEGORYID"),
           builder.count(false, "C", builder.field("COURSEID") )
 );
```

Examples of building RA expression: example 3

Schema

```
COURSE(COURSEID:int, TITLE:string, CATEGORYID:int)
CCATEGORY(CATID:int, CATNAME:string
```

Query

```
-- Show the number of courses in each course category where number of
courses is greater than 1
SELECT CATEGORYID, count(*) AS C,
FROM COURSE
GROUP BY CATID
HAVING C > 1

// Build RA expression for the above query
builder
.scan("COURSE")
.aggregate(builder.groupKey("CATEGORYID"),
           builder.count(false, "C", builder.field("COURSEID") )
 )
.filter( builder.call(SqlStdOperatorTable.GREATER_THAN,
builder.field("C"), builder.literal(1)));
```

Examples of building RA expression: example 4

Schema

```
COURSE(COURSEID:int, TITLE:string, CATEGORYID:int)
CCATEGORY(CATID:int, CATNAME:string
```

Query

```
-- Show the title of each course along with the name of the category
SELECT TITLE, CATNAME
FROM COURSE c, CCATEGORY g
WHERE c.CATEGORYID = g.CATID

// Build RA expression for the above query
builder
.scan("COURSE").as("c")
.scan("CCATEGORY").as("g")
.join(JoinRelType.INNER)
.filter( builder.equals(builder.field("c", "CATEGORYID"),
builder.field("g", "CATID")))
// Syntax:.filter (predicate1, predicate2);  where "," implies AND
.project(builder.field("TITLE"), builder.field("CATNAME"));
```

Examples of building RA expression: example 5

Schema

```
COURSE(COURSEID:int, TITLE:string, CATEGORYID:int)
CCATEGORY(CATID:int, CATNAME:string
```

Query

```
-- Show all categories from COURSE and CCATEGORY
SELECT CATEGORYID FROM COURSE
Union
SELECT CATID from CCATEGORY
```

```
// Build RA expression for the above query
builder
.scan("COURSE").project(builder.field("CATEGORYID"))
.scan("CCATEGORY").project(builder.field("CATID"))
.union(true, 1);
```

Outline

- Quick SQL review
- Query processing overview
- Relational Algebra review
- Introduction to Apache Calcite
- Join processing

Joins - multi-table queries

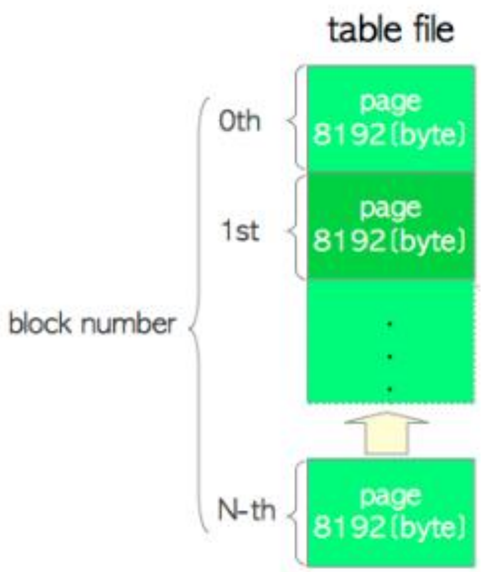
- Joins are very common
- Joins are very expensive (worst case: cross product!)
- Many approaches to reduce join cost
  - (Block) nested loops
  - Indexed nested loops
  - Sort/Merge Join
  - Hash Join
- Main goal: minimize I/O cost

ASIDE: Internal layout of a table file

- Inside the data file (heap table and index), it is divided into **pages** (or **blocks**) of fixed length, default is 8192 byte (PostgreSQL)

- Those pages within each file are numbered sequentially from 0, and such numbers are called as **block numbers**.

- If the file has been filled up, database adds a new empty page to the end of the file to increase the file size.



Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- In algebra: R $\bowtie$ S. Common! Must be carefully optimized. R x S is large; so, R x S followed by a selection is inefficient.

- Assume:
  - M pages in R, p_R tuples per page
  - N pages in S, p_S tuples per page.
  - In our examples, R is Reserves and S is Sailors.

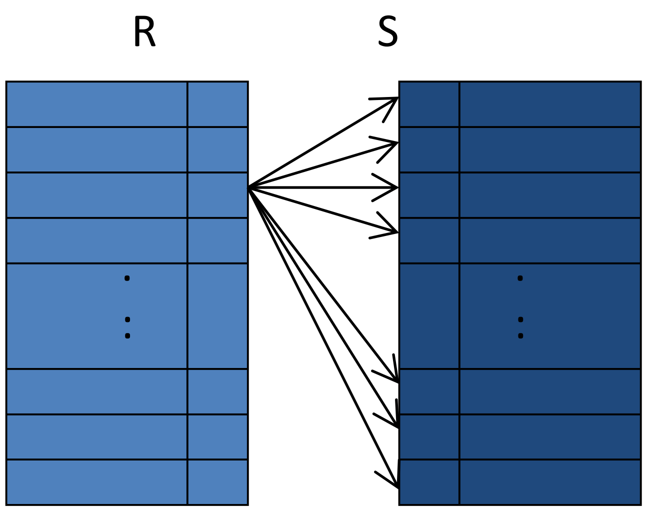
- *Cost metric* : # of I/Os. We will ignore output costs.

Nested Loop Join (NLJ)

Basic Join Algorithm

Input: Relations R and S

Output: Joined relation T



T=empty set

FOR EACH tuple r in R

FOR EACH tuple s in S

IF r.a==s.a

THEN append r||s to T

Nested Loops Join (NLJ)

```
foreach tuple r in R do
        foreach tuple s in S do
                if r_i == s_j  then add <r, s> to result
```

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.

- How much does this Cost?

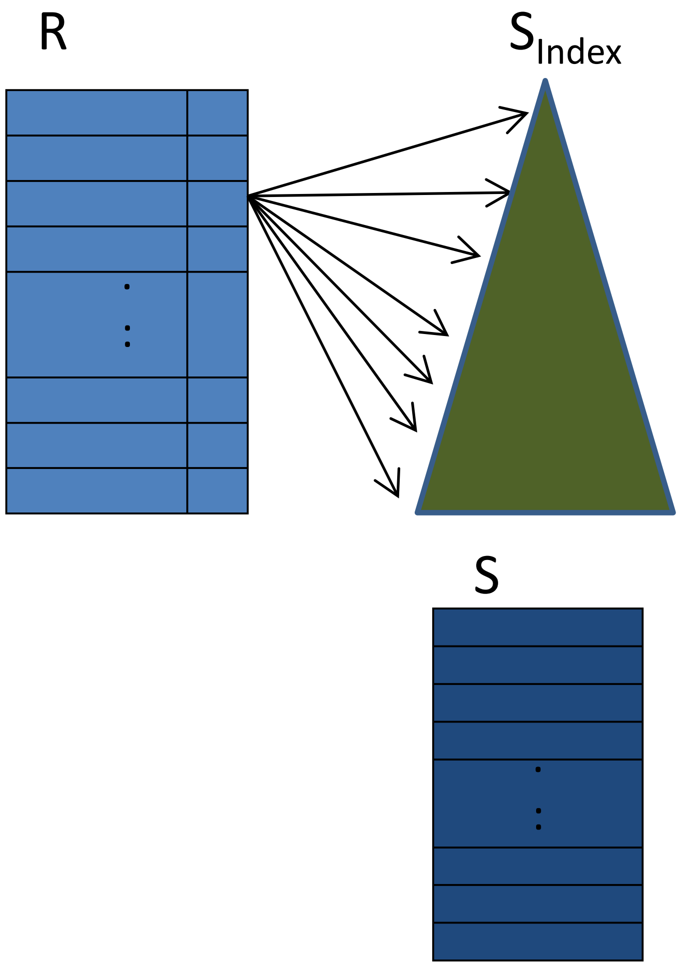
  $(p_R * M) * N + M$ = 100*1000*500 + 1000 = **50,001,000** I/Os.
  - At 10ms/IO, Total: ???

Index Nested Loop Join (INLJ)

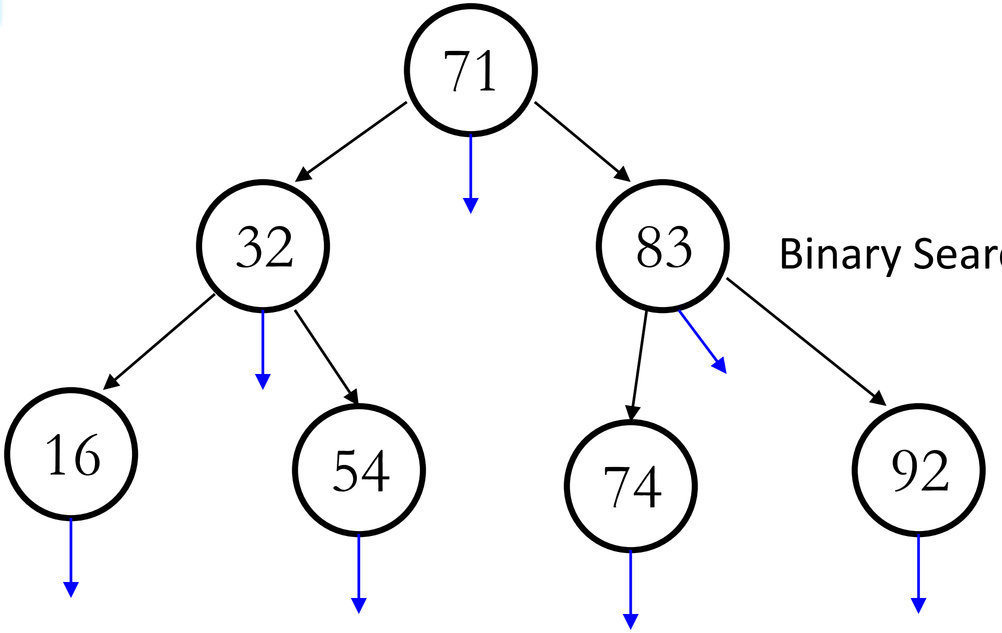
Basic Join Algorithm
Input: Relations R and S
Output: Joined relation T

T=empty set
FOR EACH tuple r in R
  FOR EACH tuple s returned by S_{index} where r.a==s.a
    Append r||s to T

Assumption: an index exists on the join column S.a



Quick review: Persistent Binary Search Tree?



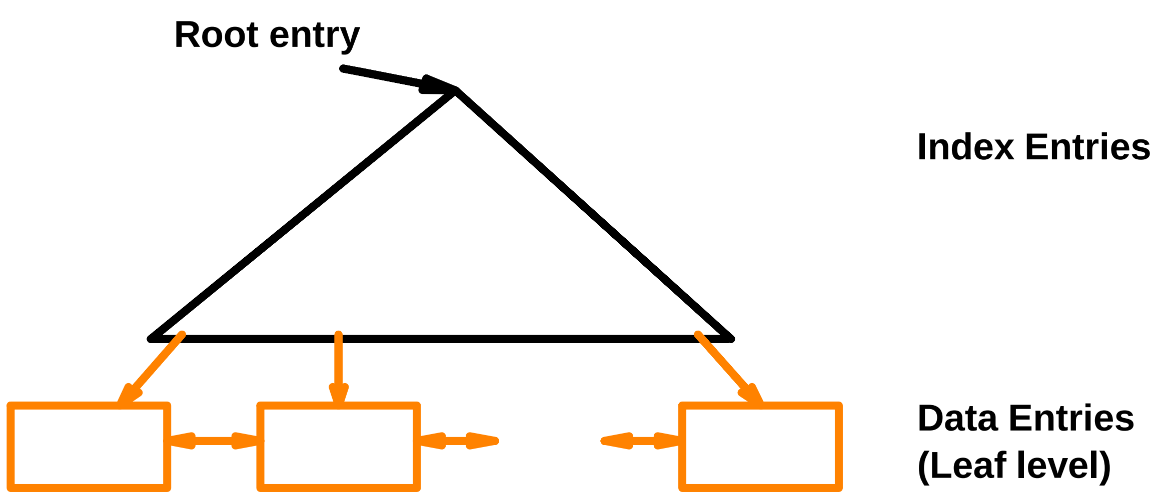
Binary Search Tree node

Each node contains one entry

Quick review: B+ Tree Index

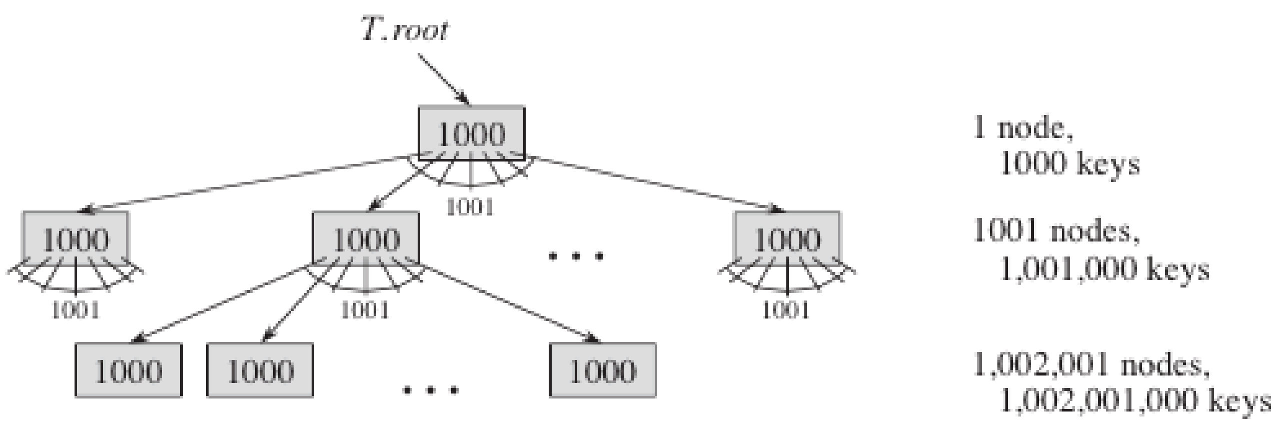
- Minimum 50% occupancy (<u>except for root</u>). Each node contains between *t* and *2t* entries. The parameter *t* is called the *order* or *minimum degree* of the tree.

- Height of a B^+-tree: $h = \log_t N$
  (N = total number of keys)



Quick review: B^+ Tree Index

- To amortize disk cost, data is transferred in large chunks.
  - Think of B-Tree as a BST with very fat nodes
  - Each node can store keys for about 50-2000 items, and will have the similar branching factor



- With a branching factor of 1001 (1000 keys per node), 1 billion keys can be accessed by a tree of height 2.
  - Just 2 disk accesses!

Index Nested Loops Join (INLJ)

```
foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result
```

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: $M + ((M*p_R) *$ cost of finding matching S tuples)
- For each tuple, cost of probing index is about 2 - 4 IOs for B^+ tree.
- For each Reserves tuple: 2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple.
  - Cost = 1000 + (100*1000)* 3 = 301,000 I/Os.
  - Cost (if, index fits in memory) = 1000 + (100*1000) = 101,000 I/Os

Other Join Approaches

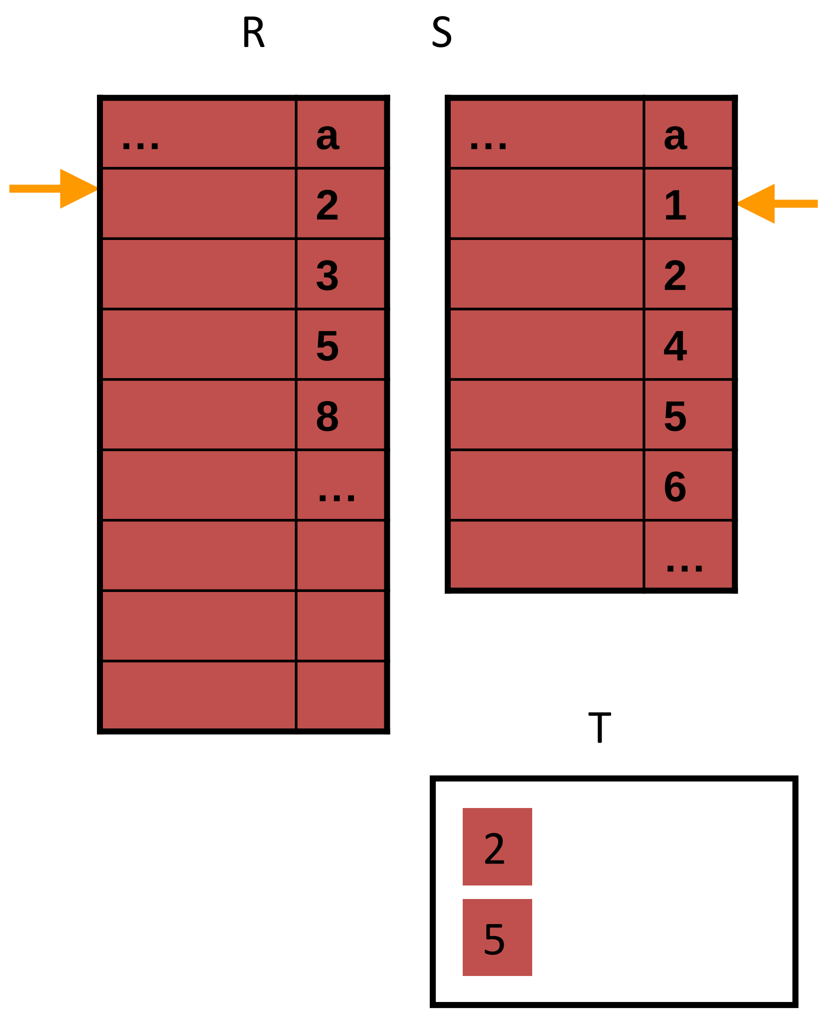
- Sort-merge join
  - Sort both relations first.
  - Scan through both relations only once.

- Hash-based join
  - No need to sort both relations first.
  - Relations have hash indices.
  - Use hash to find the corresponding records.

Sort-merge Join

Input: Sorted relations R and S
Output: Joined relation T

- General scheme:
  - Do { Advance scan of R until current R-tuple >= current S tuple;
    Advance scan of S until current S-tuple >= current R tuple; }
    Until current R tuple = current S tuple.
  - At this point, all R tuples with same value in R and all S tuples with same value in S *match*; output <r, s> for *all* pairs of such tuples.
    - Like a mini nested loops
  - Then resume scanning R and S.



Cost of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N

Cost ≈ 15,950

Hash-based Join

Input: Relations R and S

Output: Joined relation T

Build hashtable-R on relation R

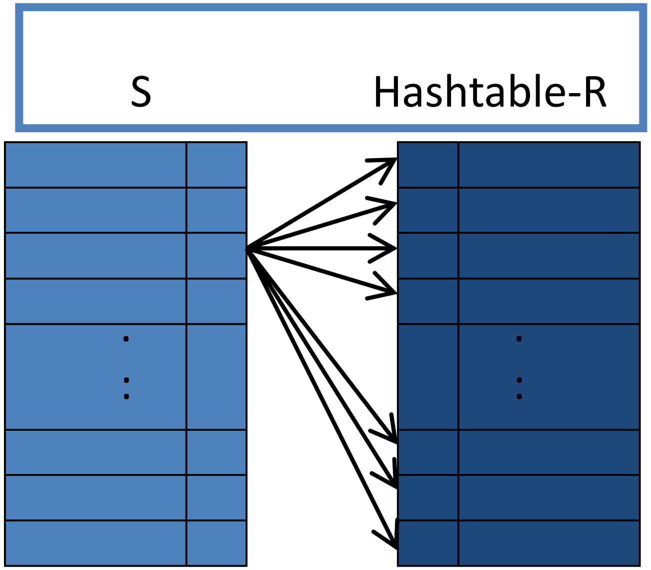
T=empty set

FOR EACH record s in S

Check hashtable-R,

IF a match found *[i.e. h(r.a)==h(s.a)]*

THEN append r||s to T



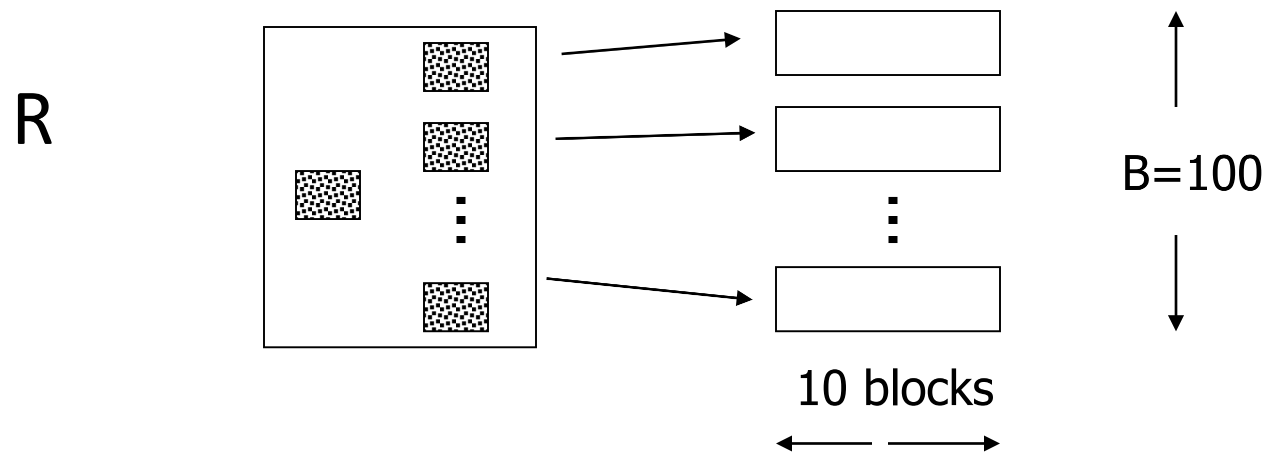
- Build in-memory hash table on R
    - R is called the build relation of the hash join
- S is called the probe relation of the hash join

Disk-based Hash Join – partitioning phase

- Relations R, S

→ Use B buckets

→Read R, hash using *h1*, + write buckets

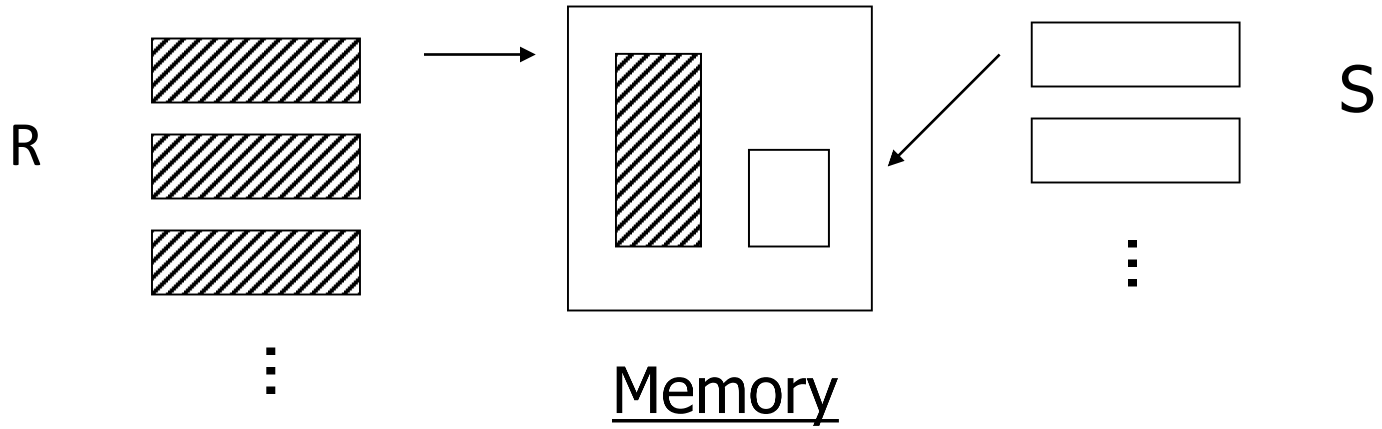


→ Same for S

Disk-based Hash Join – join phase

→ Read one R bucket; build in-memory hash table using *h2*
[R is called the build relation of the hash join]

→ Read corresponding S bucket + hash probe
[S is called the probe relation of the hash join]



Then repeat for all buckets

Cost of Hash-Join

- In partitioning phase, read+write both relns; 2(M+N).
- In Join phase, read both relns; M+N I/Os.

- In our running example, this is a total of 4500 I/Os.

Partitioning:	Read R + write
	Read S + write
Join:	Read R, S

Total cost = 3 x [1000+500] = 4500

Summary of best costs for join algorithms

Join	I/O Cost*	Time (random access)	Time (sequential access)
Nested loop join (NLJ)	50,001,000	11.57 days!	13.9 hours
Index Nested loop join INLJ	301,000 or 101,000**	33.7 minutes	1.7 minutes
Sort-merge join	15,950	5.3 minutes	16 seconds
Hash join	4500	1.5 minutes	4.5 seconds

* Note: these versions do <u>not</u> use additional in-memory buffer to cache some of the pages. With in-memory buffering, the I/O cost is be significantly improved.

** If the index fits in memory

Summary of join algorithms

- NLJ ok for "small" relations

  (when data fits in memory)

- For equi-join, where relations not sorted and no indexes exist, Hash Join usually best

Summary of join algorithms

- Sort-Merge Join good for

  non-equi-join (e.g., R1.C > R2.C)

- If relations already sorted, use

  Merge Join

- If index exists, it could be useful
  - Depends on expected result size and index clustering

- Join techniques apply to Union, Intersection, Difference

Summary

- A virtue of relational DBMSs: *queries are composed of a few basic operators*; the implementation of these operators can be carefully tuned (and it is important to do this!).
- Many alternative implementation techniques for each operator; no universally superior technique for most operators.
- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.

Next

- Parallel Databases..