CS448/648 Database Systems Implementation

Tutorial 1: Internals of PostgreSQL

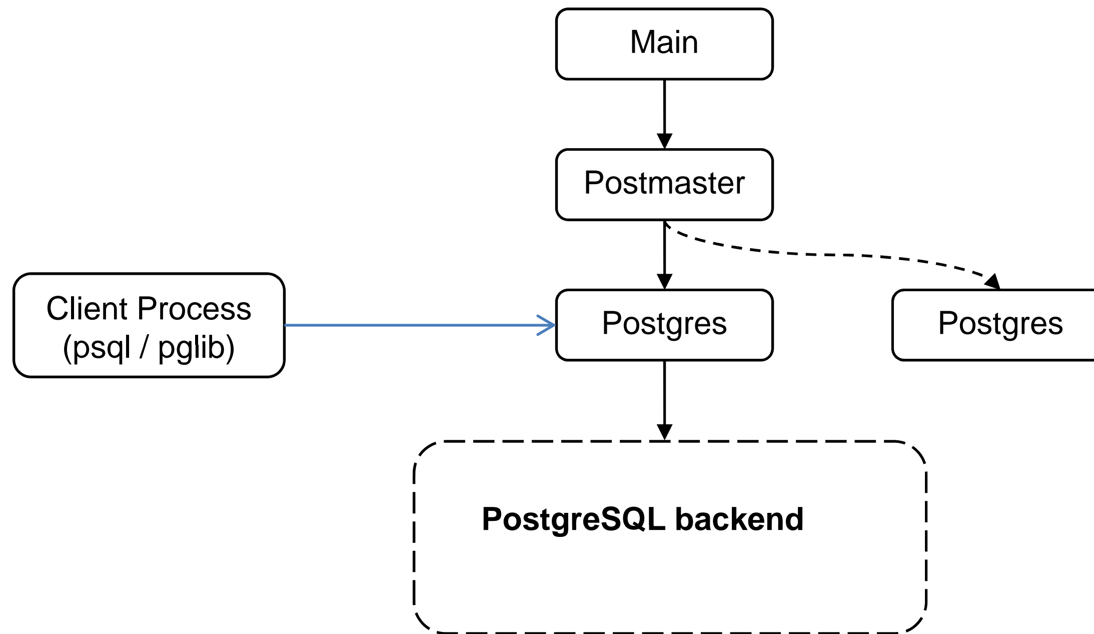
Outline

1. Introduction to PostgreSQL
2. PostgreSQL Architecture
3. PostgreSQL Components
   – Parser
   – Query Rewriter
   – Optimizer
   – Executor
4. Symmetric Hash Join

Introduction to PostgreSQL

- PostgreSQL is an open-source, object-relational database system.
- PostgreSQL was first developed at University of California, Berkeley under the name POSTGRES.
- Throughout this course, we will use version 8.1.4 as a code-base to implement new features on top of it.

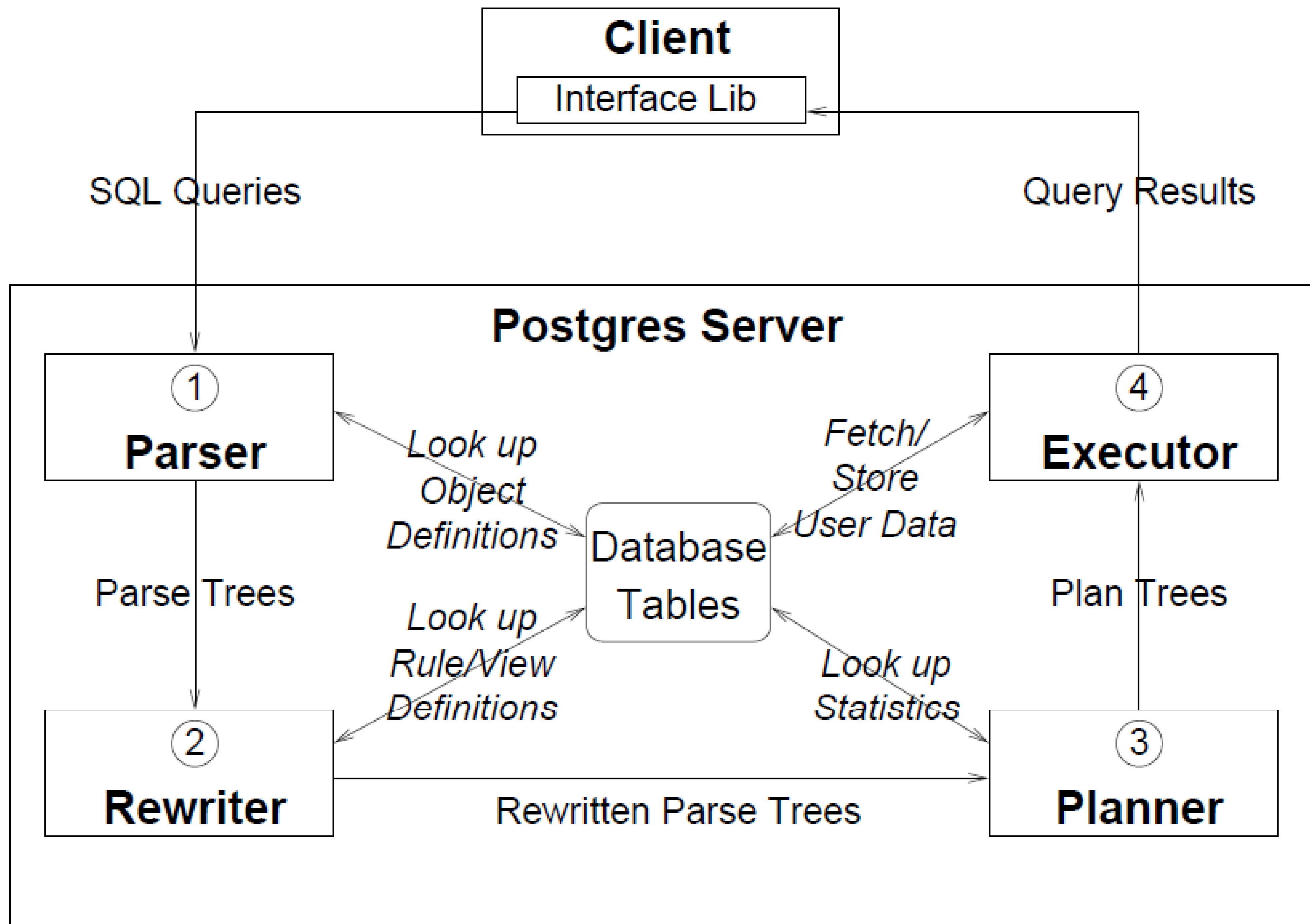
PostgreSQL Architecture



Types of Clients

- **psql**
  - Psql is an interactive client that allows the user to submit SQL queries.
- **libpq**
  - Libpq is the C application programmer's interface (API) to PostgreSQL. libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server.
- **Server Programming Interface (SPI)**
  - SPI gives writers of user-defined C functions the ability to run SQL commands inside their functions. SPI is a set of interface functions to simplify access to the parser, planner, optimizer, and executor. SPI also does some memory management.

PotgreSQL Backend



Reference: Tom Lane, A Tour of PostgreSQL Internals

Query Parser

- SQL query is tokenized and parsed according to SQL language standard.
- It parses and analyzes the string input and gives out a **Query** structure (Query Tree) for the executor.
- Syntax-errors are caught at this stage.
- Source code located in the directory

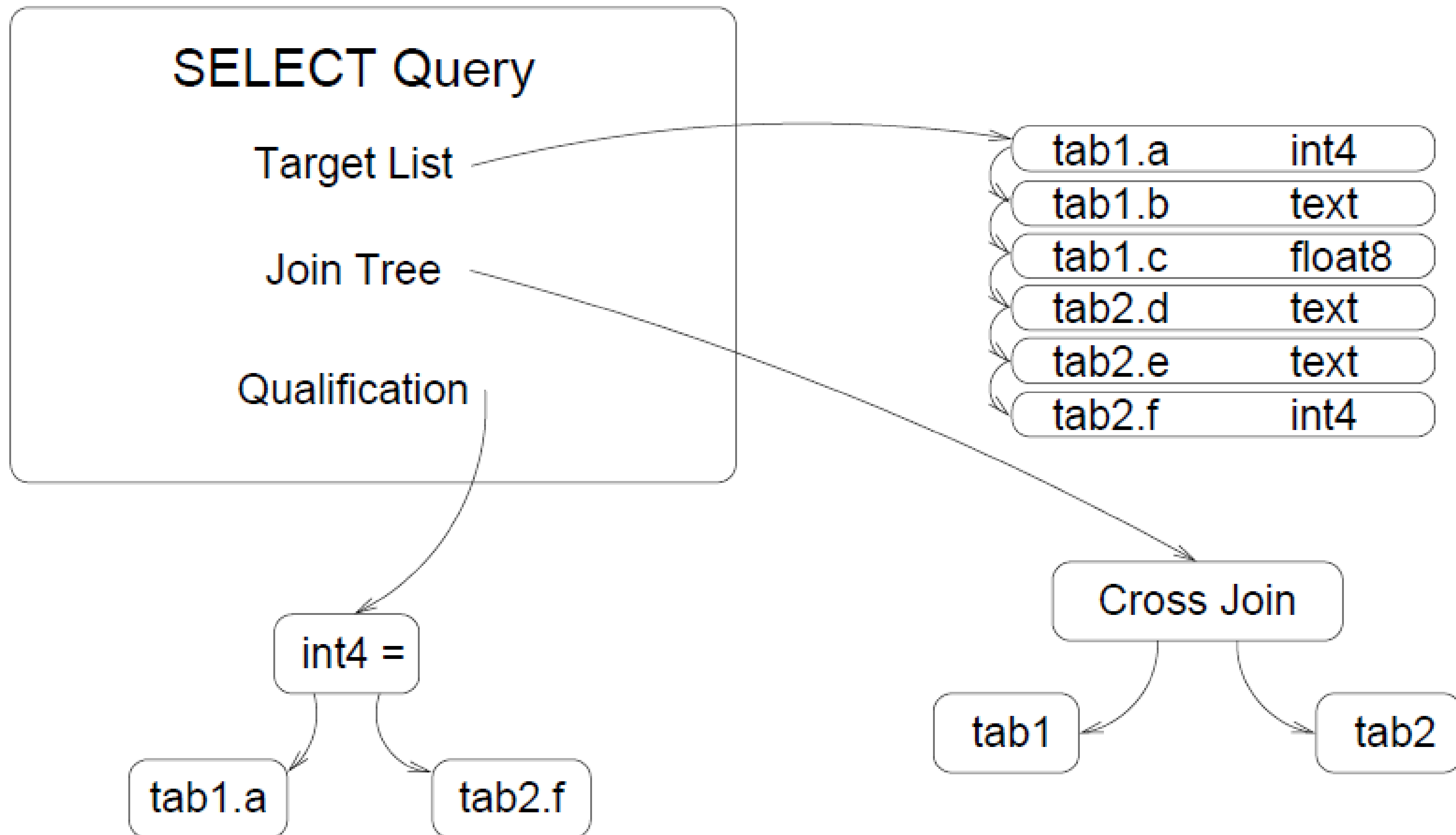
  src/backend/parser

Example

Input:

```
SELECT * FROM tab1, tab2 WHERE tab1.a = tab2.f
```

Output:



Reference: Tom Lane, A Tour of PostgreSQL Internals

Query Rewriter

- Also called the *Rule System*.
- It modifies the Query structure based on a set of rules before passing it to the optimizer.
- Rules can be user-defined or automatically created for views.
- Rules types are ON SELECT, ON UPDATE/INSERT/DELETE

- Example :

  SELECT * FROM Tab1, **View2** ….
  is flattened to
  SELECT * FROM Tab1, **(SELECT * FROM Tab2,Tab3 ….) AS View2** …

Query Rewriter: Example

- Say we want to trace changes to the **sl_avail** column in the **shoelace_data** relation in a log table:

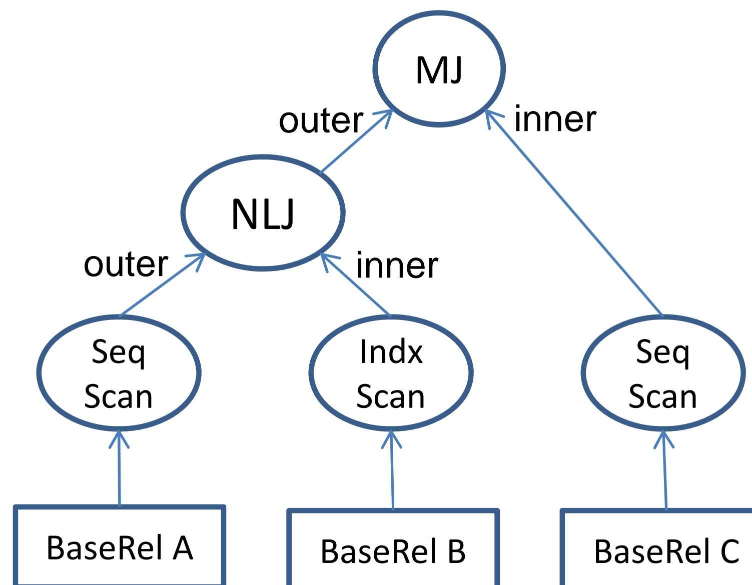
```
CREATE RULE log_shoelace
AS ON UPDATE TO shoelace_data
WHERE NEW.sl_avail <> OLD.sl_avail
      DO INSERT INTO shoelace_log
      VALUES ( NEW.sl_name,
                      NEW.sl_avail,
                   current_user,
                   current_timestamp );
```

PostgreSQL Optimizer

- PostgreSQL uses bottom-up optimization (dynamic programming).
- Optimizer accepts a **Query** structure and produces a plan with the least estimated cost.

Path Structure

- Possible physical plans to answer the query are stored in a structure named **Path** .
- A Path is a hierarchical structure. Each node represents a query operator.
- A Path specifies the access methods, the join order and the join algorithms used at each node.
- Example path:

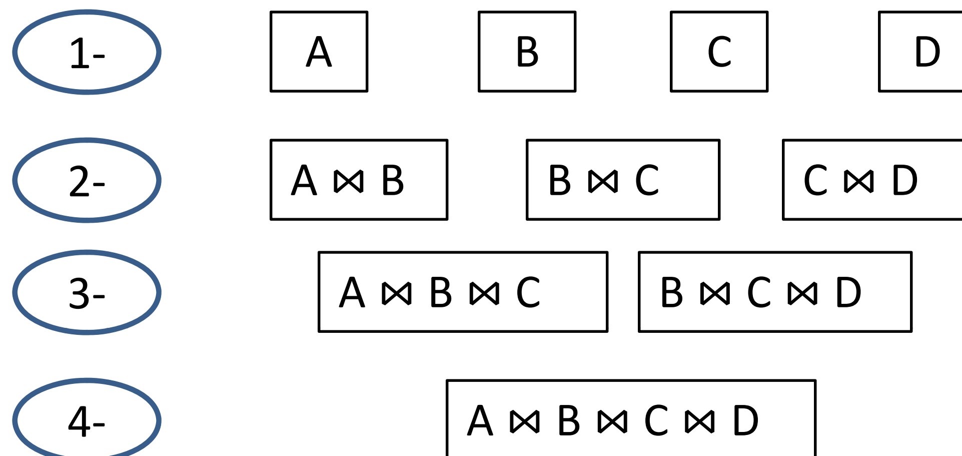


Query Operators

- Unary operators : accepts one input relation
  - Sequential Scan/ Index Scan
  - Sort
  - Unique
  - Aggregate
  - Materialize
- Binary operators : accepts two input relations
  - Nested Loop Join
  - Hash Join
  - Merge Join

Constructing Paths

- Paths are constructed in a bottom-up style.
- Two main types of relations:
  - Base Rel : could be primitive tables, or subqueries that are planned via a separate recursive invocation of the planner
  - Join Rel : is a combination of base rels
- Joinrels are constructed incrementally. Larger joinrels are constructed by combining smaller baserels and joinrels.
- Example : constructing $A \bowtie B \bowtie C \bowtie D$



Entry Point and Important Files

- Optimizer component is included in the directory : **src/backend/optimizer**
- The optimizer entry point is in the file

  src/backend/optimizer/plan/planner.c
- Path construction and cost estimation is included in the directory

  src/backend/optimizer/path
- A **README** file is included in the optimizer directory for more details.
- You can use the **Explain** command to print the selected plan, along with estimated and actual statistics (e.g. cardinality, execution time)

Plan Executor

- After finding the path with the least cost, a **Plan** is constructed from the found path.
- There is a one-to-one mapping between a Path and a Plan. Different information are kept in each structure that suits the query processing stage (optimization / execution)
- Similar to a the Path structure, a Plan is a hierarchical structure of query operators.

Query Operators

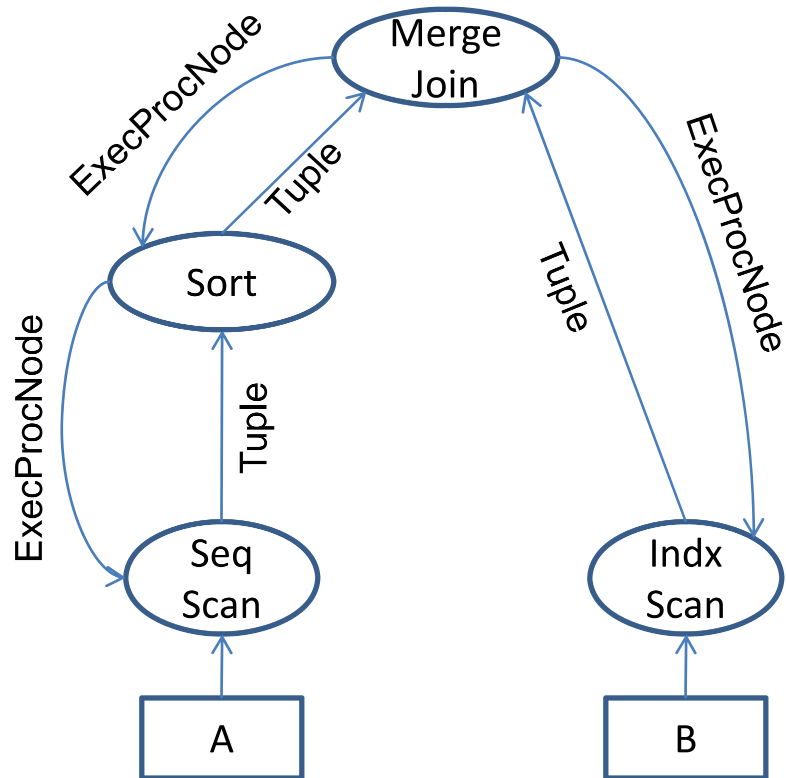
- Operators can be classified to two categories:
  - Blocking Operators : sort , aggregate
  - Non-blocking operators (pipelined) : Index Scan, Merge join , Nested-Loop Join
- Pipelining tuples through operators allows fast reporting of results; user does not have to wait till all of the input tuples are processed before start getting results.

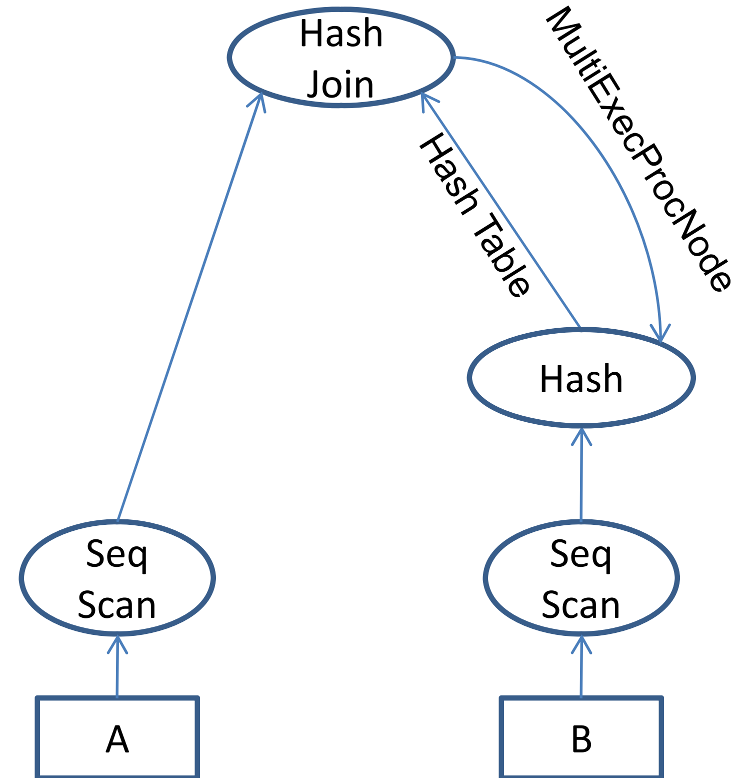
Tuples Retrieval

- Executor is based on demand-pull interface.
- There are mainly two types of Plan node execution: *single-tuple retrieval* and *multiple tuples retrieval*.

- single-tuple retrieval
  - At each node in the plan, a tuple is requested from the children nodes through a call to the function **ExecProcNode**. After processing input tuples, one output tuple is constructed and returned to the caller.
  - Examples: scan, join, sort
- multiple tuples retrieval
  - If a node does not support one-by-one retrieval, all tuples are processed and returned to the caller in some structure (e.g. hash table or a bitmap table). In this case , caller use **MultiExecProcNode** interface.
  - Examples: bitmap scan, hash

Example





Node Execution State

- ExecProcNode is reentrant procedure. The state of the previous execution (e.g. which tuples are already retrieved) must be stored.
- Each Plan node has a corresponding PlanState to store execution state. (e.g. Hash and HashState structures).

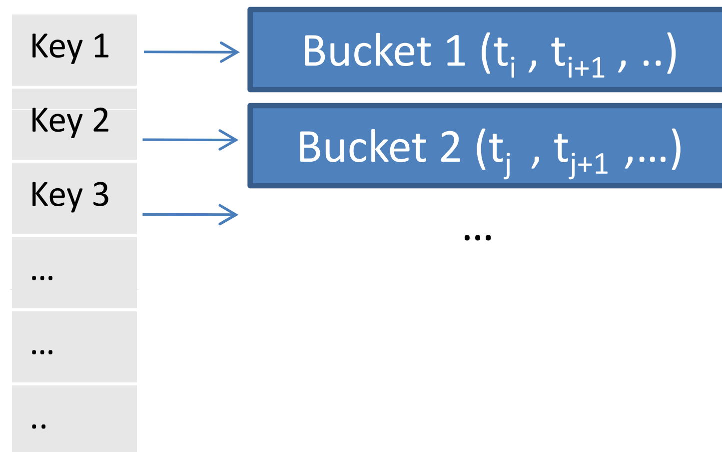
Symmetric Hash Join

- Hash Join Algorithm
- Hybrid Hash Join (current implementation in PostgreSQL)
- Symmetric Hash Join

Hash Join

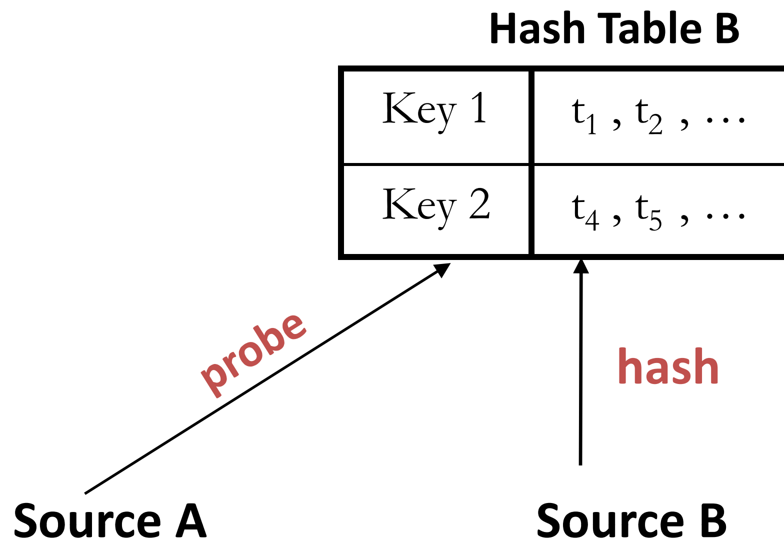
- Hash Join can be used when the join condition involves equality predicates only.
- It does not require sorted inputs. However, it requires one or more input relations to be hashed.
- Any hash join algorithm is based on two operations:
  - Hash table construction
  - Hash table probing

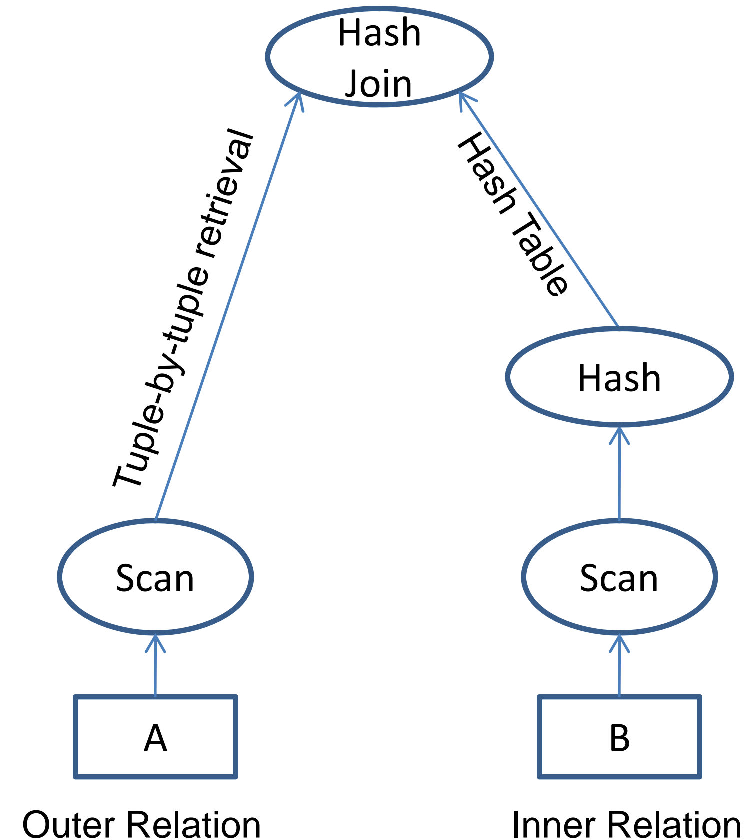
- A hashing function is used to map each key value of a tuple to a bucket number.
- The tuple is stored in the resulting bucket.
- Each bucket could contain **zero or more** tuples.



Hash Join Algorithm

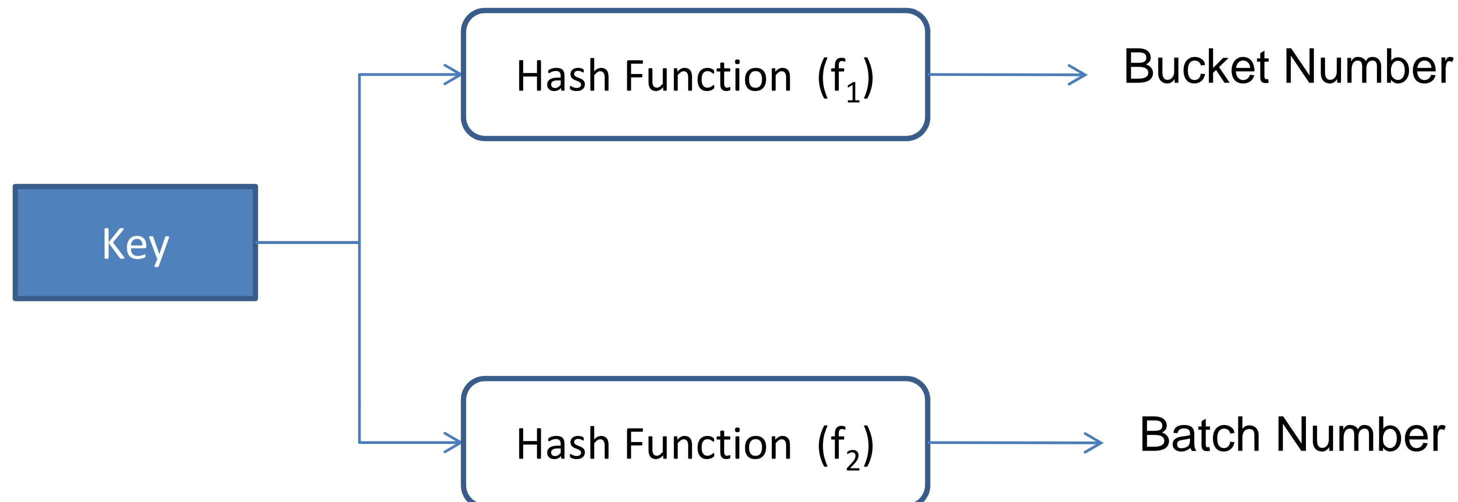
1. Build a hash table that contains all tuples from inner relation.
2. For each tuple '*t*' in the outer relation:
   a. probe the hash table using the hash value of *t*
   b. If a match found, return it





Hybrid Hash Join

- Implemented in PostgreSQL 8.1.4.
- Addresses the problem of insufficient memory to keep the hash table.
- The hash table is divided into “batches” based on a secondary hash function.
- Only one batch resides in the memory at a time.

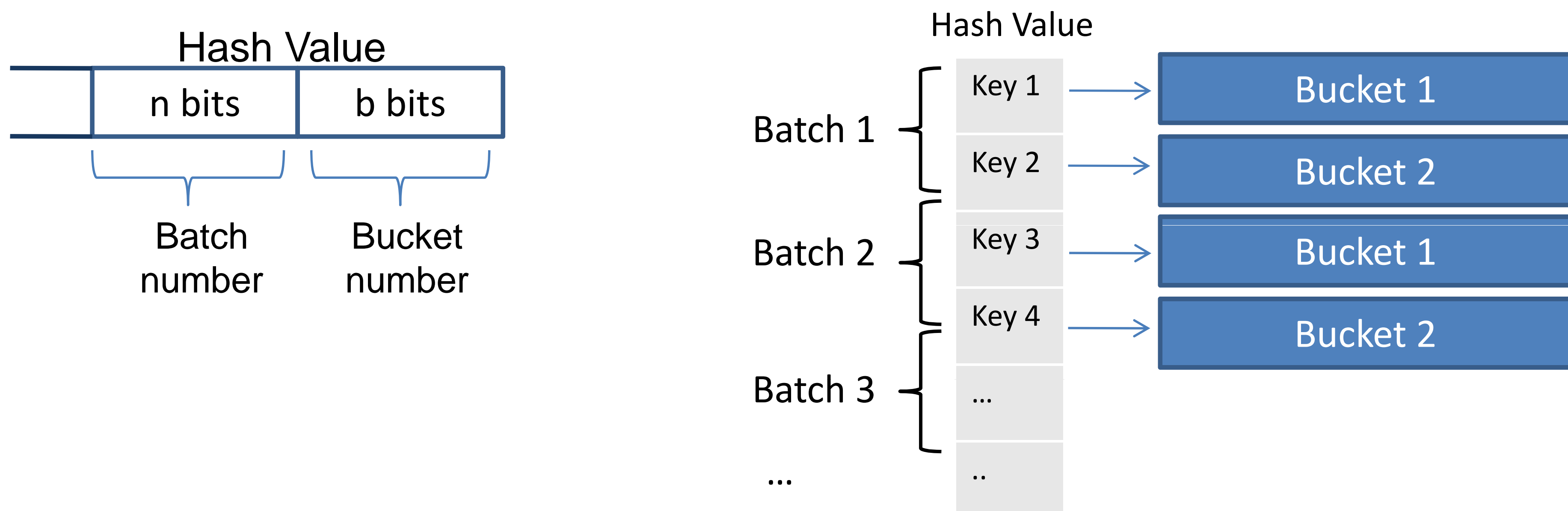


Hybrid Hash Join

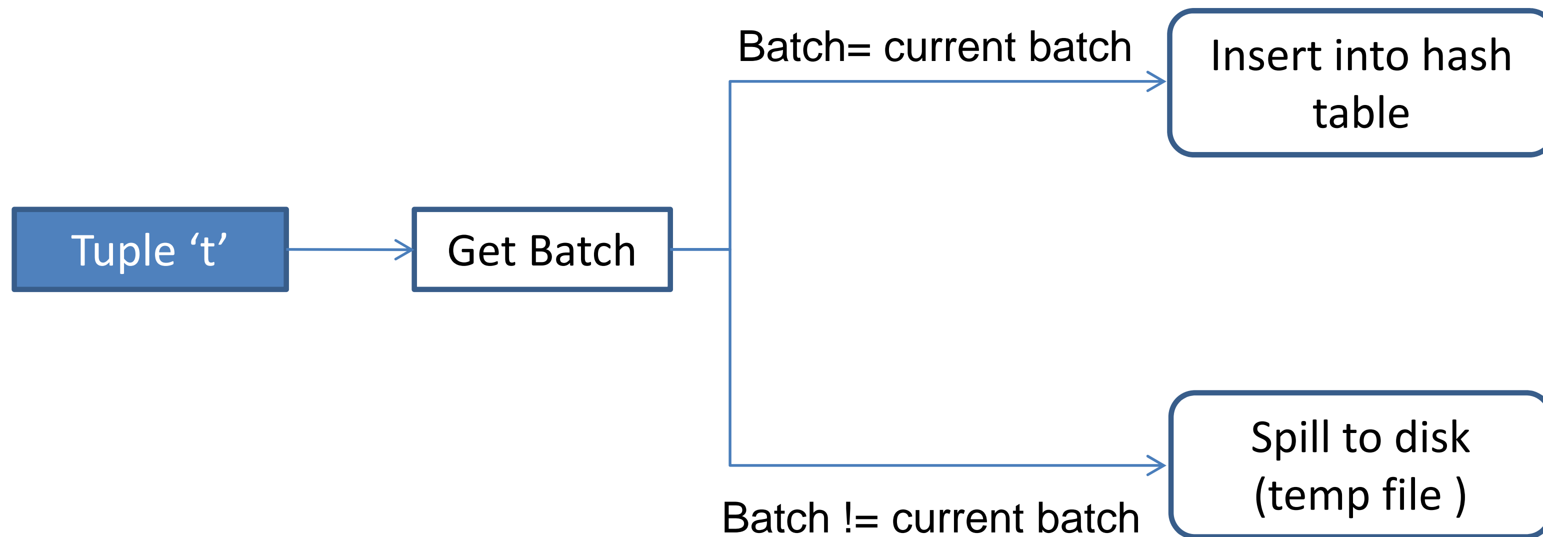
Example : Assume we need to divide tuples into **B = 2^b^** buckets and **N = 2^n^** batches using one hashing function f_1.

$$bucket = f_1(Key) \ MOD \ B$$

$$batch = [\, f_1 (Key) / B \,] \ MOD \ N$$

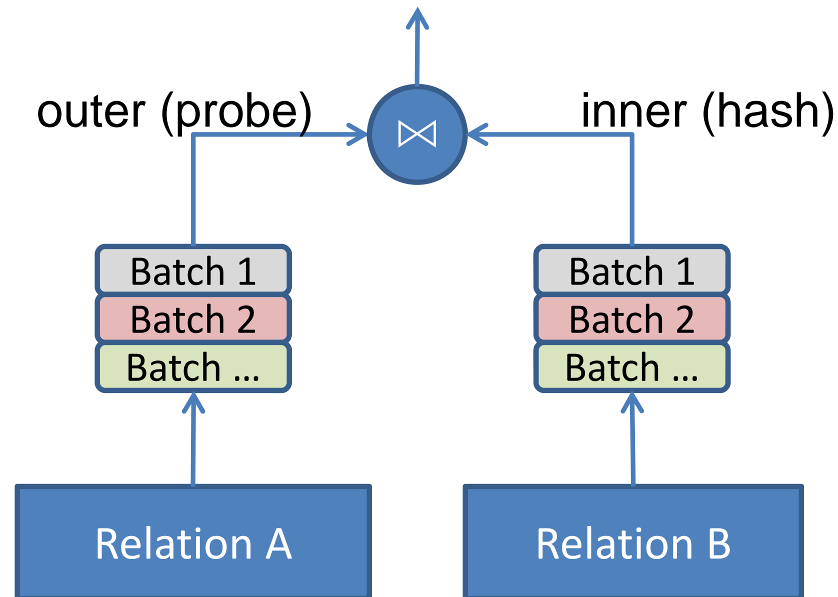


- There is only one batch in memory at one time. We call it the “current batch”.



- Disk-resident batches are retrieved in order. After processing each batch, we discard it.

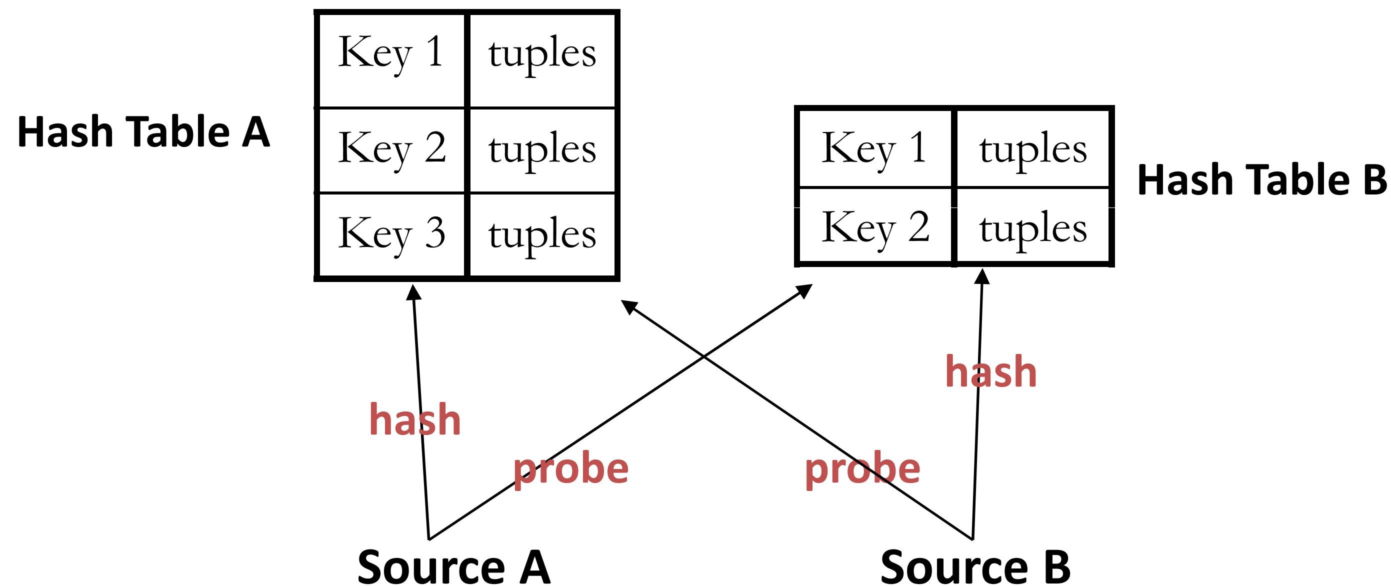
- The join algorithm is modified to handle multiple batches as follows:



1. Build a hash table that contains all tuples from inner relation. Keep the first batch in memory and spill remaining batches to disk.
2. For each tuple in the outer relation:
   a. If hash value of tuple *t* belong to the "current batch", probe the hash table and if a match found, return it.
   b. Else, spill to corresponding batch on disk.
3. Load successive batches from inner and outer and process their tuples.

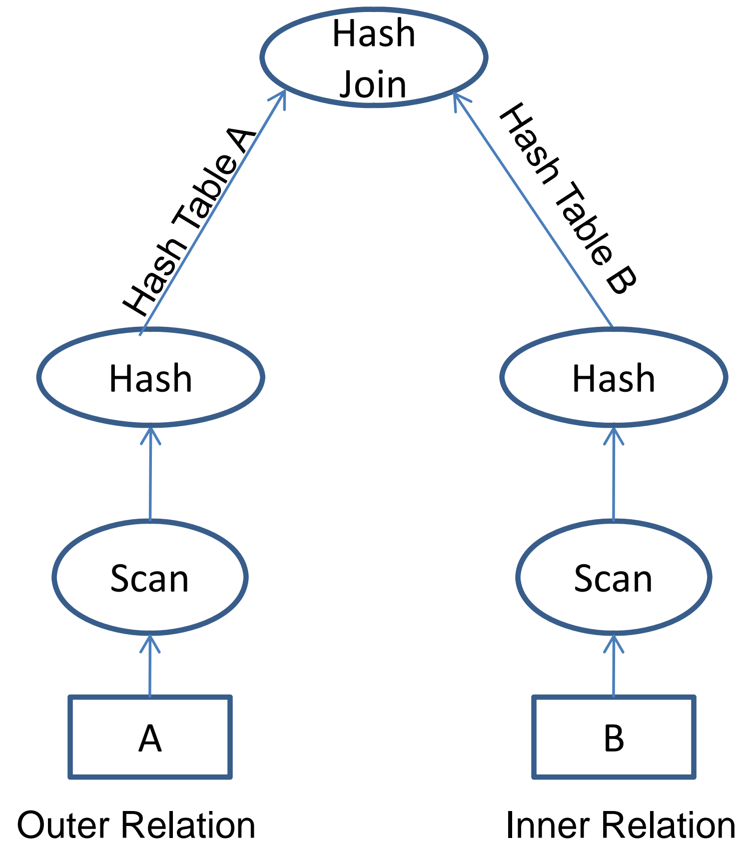
A2: Symmetric Hash Join

- Avoids the “blocking” effect of the hash table.
- Creates two hash tables, one for each relation.
- We assume in this assignment that the hash tables fit into the memory.



Symmetric Hash Join Plan

1. Get one tuple from A and insert it into Hash Table A.
2. Use tuple obtained from A to probe Hash table B. If match found return it.
3. Get one tuple from B and insert it into Hash Table B.
4. Use tuple obtained from B to probe Hash table A. If match found return it.
5. Stop when all tuples from A and B are consumed.



Implementing Symmetric Hash Join

Tasks:

- Insert additional hash node on top of outer relation.
- Modify the hash operator to support single-tuple retrieval.
- Modify execution state to keep additional information (e.g. current tuple in inner relation).
- Modify the hash join implementation to use the symmetric hash join algorithm.

Questions?