Query Processing (II)

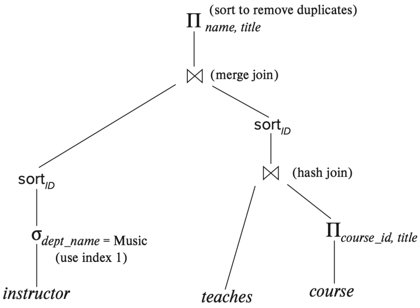
Spring, 2024

Announcements

- We are recruiting new members for our research group.
- If you are interested in joining, please drop me an email.
- Contact: `q.yin@sjtu.edu.cn`

DBMS: Operator execution

Purpose:
Execute a dataflow by operation on tuples and files.



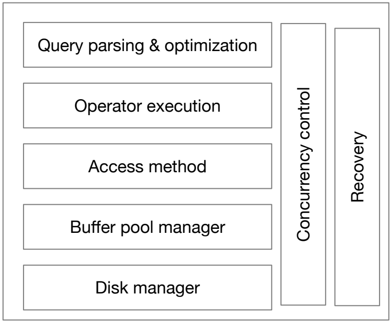


Figure: DBMS architecture

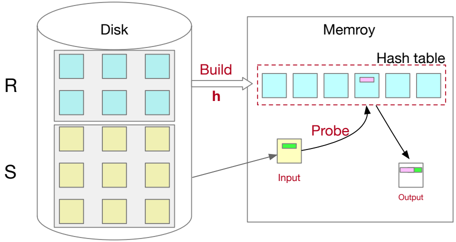
Recap

- Tables: R, S
- Tuples: t_r, t_s
- Number of tuples: $|R|$, $|S|$
- Number of pages: $P(R)$, $P(S)$
- Number of available buffer pool pages: B
- Cost metric: number of I/O's

Hash join

- Applicable for equi-joins and natural joins, e.g., $R \bowtie_{R.A=S.B} S$.
- If $t_1 \in R$ and $t_2 \in S$ can join, then they have the same value on the join attributes.
- Use a hash function h to partition both relations.
- Compute the join results on each partition.

Basic in-memory hash join



- Build phase: scan the outer table R and construct a hash table using a hash function h on the join attributes.
- Probe phase: scan the inner table S and use h on each tuple $t \in S$ to jump to the location in the hash table and find a matching tuple.
- Cost: $P(R) + P(S)$.
- Buffer pool requirement: $B \geqslant P(R) + 2$ or roughly the outer table R can fit in memory.

Hash join: partition phase

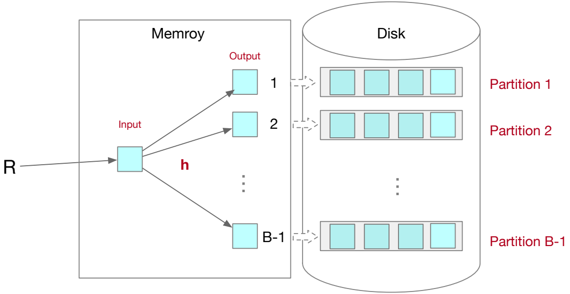
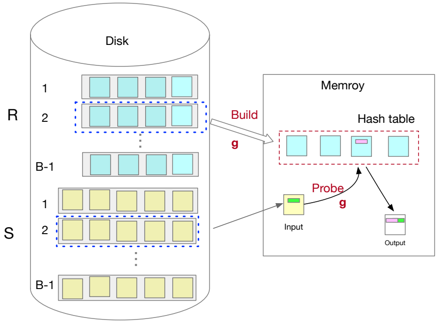


Figure: Partition R with h (need to do the same for S)

- Partition both R into $B-1$ partitions, using a hash function h on the join attributes.
- A buffer block/page is reserved as the output buffer for each partition.
- Partition table S in the same way.

Hash join: build & probe phase



- Read each partition R_i of R and build a hash table using another hash function g.
  – The hash functions g and h must be different. Why?
- Read the corresponding partition S_i of S in a per-page basis; then probe and join.
- R is the build relation and S is the probe relation.

Cost analysis

Assumption

- Partition phase divides table R into $(B-1)$ partitions evenly. That is, each partition of R has $\lceil P(R)/B-1 \rceil$ pages.
- Build & probe requires $\lceil P(R)/B-1 \rceil \leqslant B-2$, i.e., every partition of R fits into memory.
- $P(R) \leqslant (B-1)(B-2) \approx B^2$. Thus roughly $B \geqslant \sqrt{P(R)}$.
- We have no size requirement for the probe relation S.
  — Use the smaller input as the build relation R.

Cost: $3(P(R)+P(S))$

Question. What if a partition of R is too large for memory?

Hash-based algorithms

- Union, intersection, difference.
  – More or less like hash join.
- Duplicate elimination.
  – Eliminate duplicates within each partition.
- Group by aggregation.

  – (i) Apply the hash functions to the group-by columns.
  – (ii) Tuples in the same group will end up in the same partition.

Indexed nested loop join

1. for each tuple t_r in R do
2. for each tuple t_s in Index$(t_r.A)$ do
3. add $t_r \bowtie t_s$ to the result

Figure: Algorithm for $R \bowtie_{R.A=S.B} S$, using an index of S on attribute B

- Idea: use a value of $R.A$ to probe the index on $S.B$.
- Cost analysis: $P(R) + |R| * C$.
- C is the I/O cost of an index lookup, which is $2 \sim 4$ I/O's typically.
- If both R and S support index lookup, better pick the smaller one as the outer relation.

Join algorithms (recap)

Algorithms	I/O costs
Naive Nested Loop Join	$P(R) + \|R\| * P(S)$
Block Nested Loop Join	$P(R) + P(R) * P(S)$
Indexed Nested Loop Join	$P(R) + \|R\| * C$
Merge Join	$P(R) + P(S)$
In-memory Hash Join	$P(R) + P(S)$
Hash Join	$3 * (P(R) + P(S))$

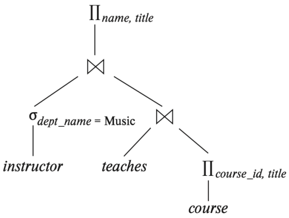
Table: Algorithms for $R \bowtie S$

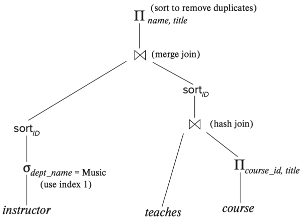
See some examples of query processing here.

Query Processing Model

Query processing overview

```sql
SELECT name, title
FROM instructor natural join teaches
     natural join course
WHERE dept_name ='Music';
```





- Each node of a logical plan is a relational operator.
- Each node of a physical plan represents an operator algorithm.
- Data flows from the leaves of the physical plan tree up towards the root.

Processing model

A DBMS's processing model defines how the system executes a physical query plan.

Materialization Model

- Compute the tree bottom-up.
- Children write intermediate results to temporary files.
- Parents read temporary files.

Iterator Model

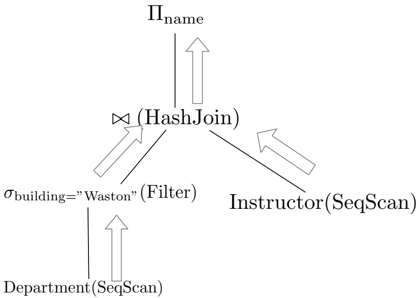
- Do not materialize intermediate results.
- Children pipeline their results to parents.
- Also known as volcano model or pipeline model.

Materialization model

- Evaluate one operator at a time, starting at the leaves.
- Use intermediate results materialized into temporary relations to evaluate next-level operators.

Example.

```
SELECT name
FROM department NATURAL JOIN instructor
WHERE department.building="Watson"
```



- Good for queries that touches a few records at a time, e.g., OLTP workload.
- Not good for OLAP queries with large intermediate results.

Iterator model

Every operator maintains its own execution state and implements a next_tuple method.

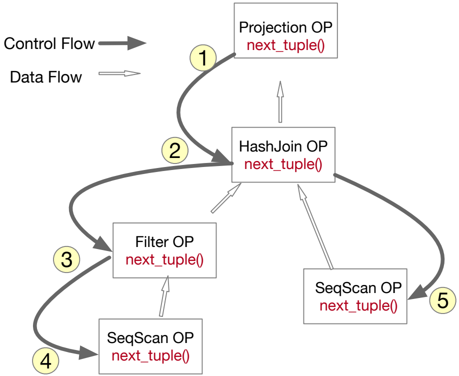
```
class Operator {
public:
  virtual Status init() = 0;
  virtual Status next_tuple(Tuple &tuple) = 0;
};
```

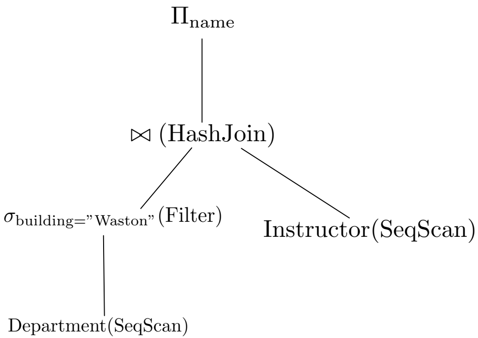
Figure: Operator Iterator Interface

One each invocation, the operator

- Return the next tuple in the result
- Or return a null pointer if there are no more tuples.
- Adjust state to allow subsequent tuples to be obtained.

Iterator model example: pull-based execution





- Call next_tuple() repeatedly on the root
- Iterators recursively call next_tuple() on the inputs.

Iterator model example (1): SeqScan Operator

```
class SeqScanOperator : public Operator {
public:
  SeqScanOperator(Table *table) : table(table) {}
  Status init() override {
    iter = table->begin();
    return Status::InitOk;
  }
  Status next_tuple(Tuple &tuple) override {
    if (iter != table->end()) {
      tuple = iter.get_tuple();
      iter = iter.forward();
      return Status::HaveMoreOutput;
    }
    return Status::Finished;
  }

private:
  Table *table;
  TableIterator iter;
};
```

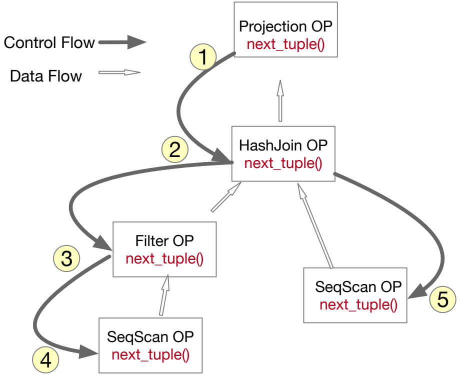
Iterator model example (2): Filter Operator

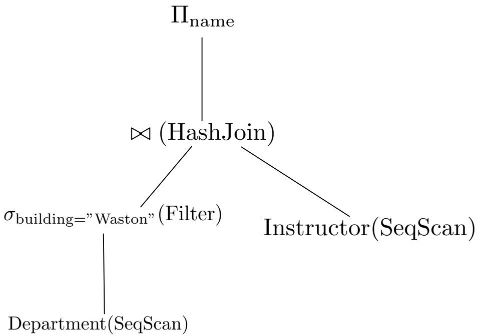
```
class FilterOperator : public Operator {
public:
  FilterOperator(Operator *child, Expression *predicate)
      : child(child), predicate(predicate) {}
  Status init() override { return child->init(); }
  Status next_tuple(Tuple &tuple) override {
    Status status;
    Tuple child_tuple;
    while ((status = child->next_tuple(child_tuple)) ==
           Status::HaveMoreOutput) {
      if (predicate->eval(child_tuple) == BooleanValue::True()) {
        tuple = child_tuple;
        return Status::HaveMoreOutput;
      }
    }
    return status;
  }
  ...
};
```

Iterator model example (3): HashJoin Operator

```
Status next_tuple(Tuple &tuple) override {
    while (true) {
      switch (state) {
      case HashJoinState::Build:
        // TODO: use the left table to build a hash table
        state = HashJoinState::ProbeRight;
        break;
      case HashJoinState::ProbeRight:
        // TODO: use the left table to probe
        if (status != Status::HaveMoreOutput) { return status; }
        break;
      case HashJoinState::MatchLeft:
        // TODO: join
        state = HashJoinState::ProbeRight;
        break;
      }
    }
}
```

Iterator model example: recap





- Pull-based execution: (i) Call next_tuple() repeatedly on the root; (ii) Iterators recursively call next_tuple() on the inputs.
- Some operators have to block until their children emit all of their tuples, e.g., Joins, Sort.

See here for more sample codes.

Vectorization model

Like the iterator model, every operator maintains its own execution state and implements a next_chuck method.

```
class Operator {
public:
  virtual Status init() = 0;
  // A DataChunk contains multiple arrays (i.e. column segments)
  virtual Status next_chunk(DataChunk &chunk) = 0;
};
```

- Each invocation emits a batch of tuples instead of a single tuple.
- Ideal for OLAP workloads since it greatly reduces the number of invocations per operator.
- Allows for operators to use vectorized (SIMD) instructions to process batches of tuples.

See here for more sample codes.