Query Processing

Fall, 2025

DBMS: Disk manager

Manage the database in files on disk.



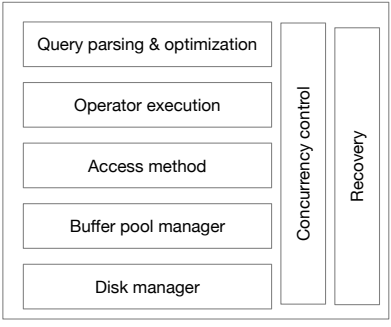
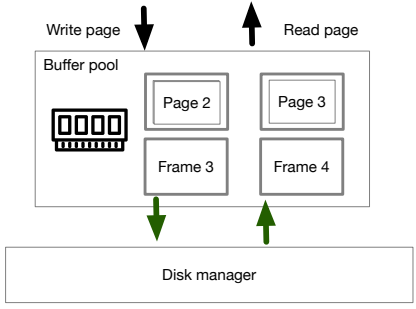


Figure: DBMS architecture

DBMS: Buffer pool manager

Provide the illusion of DBMS operating directly in RAM.



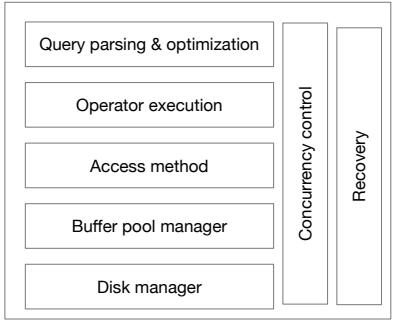


Figure: DBMS architecture

DBMS: Access method

Enables the DBMS execution engine to read and write data from pages more efficiently.

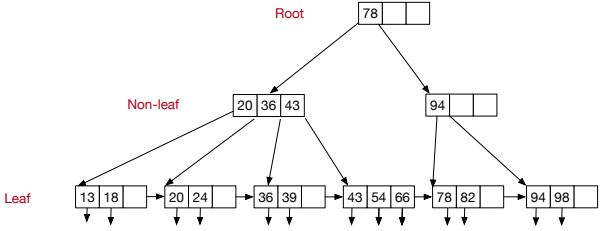


Figure: A sample B^+-tree index

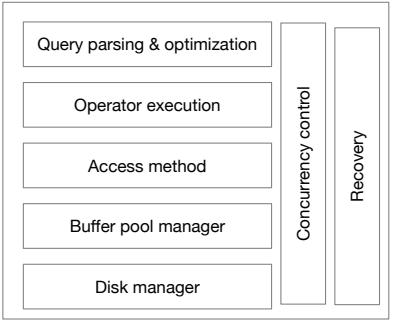


Figure: DBMS architecture

What's next?

- Query parsing and optimization: Translate SQL query into an optimized physical plan.
- Operation execution: Execute a dataflow by operation on tuples and files.

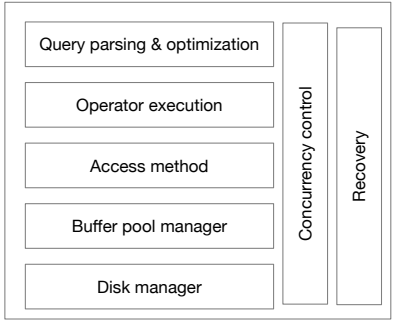
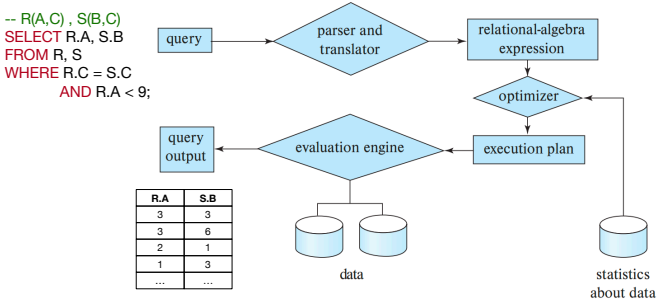


Figure: DBMS architecture

Query Processing Overview

Query processing overview



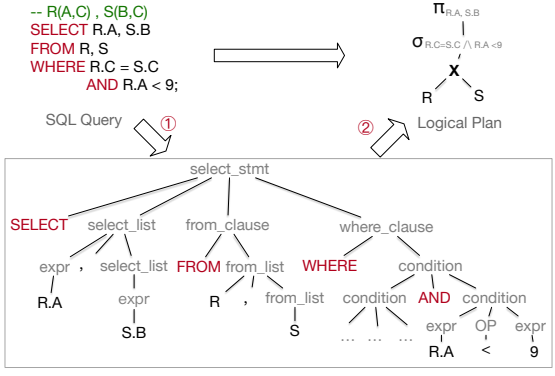
1. SQL parsing and translation: Translates the SQL query into an initial logical plan.
2. Query optimization: Transforms the logical plan into an optimized physical plan.
3. Query execution: Executes the physical plan over the database to produce query results.

Parsing and translation

The DBMS converts an SQL query into an initial logical query plan.

1. SQL parsing: The DBMS checks the syntax and semantics of the SQL statement and builds a parse tree representing its structure.

2. Analysis and translation: The parse tree is analyzed and then translated into a logical plan — a tree of relational operators that captures the query's semantics.



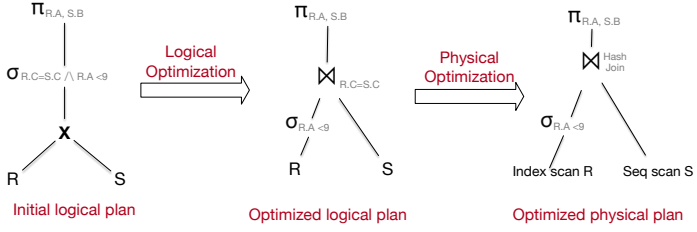
SQL Parse Tree

Query Optimization

The DBMS transforms the initial logical plan into an optimized physical plan.

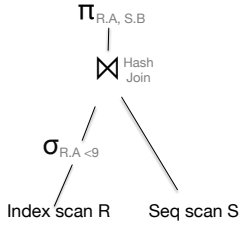
Query optimization generally consists of two main phases:

- Logical optimization: Rewrites the logical plan into an equivalent but more efficient form using RA equivalence rules.
- Physical optimization: Selects the most efficient algorithms and access paths to implement each operator in the plan.



Query execution

The DBMS executes the optimized physical plan to produce the final query result.



R.A	S.B
3	3
3	6
2	1
1	3
...	...

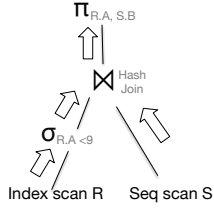
Common query execution models include:

- Materialization model: Each operator fully computes and stores its output before passing it to the next operator.
- Iterator (Volcano) model: Each operator provides `init()` and `next_tuple()` interfaces; tuples are pulled from children on demand.

Materialization model

- Evaluate one operator at a time, starting at the leaves.
- Use intermediate results materialized into temporary relations to evaluate next-level operators.
- Good for OLTP queries that touches a few records at a time.
- Not good for OLAP queries with large intermediate results.

```
-- R(A,C) , S(B,C)
SELECT R.A, S.B
FROM R, S
WHERE R.C = S.C
        AND R.A < 9;
```



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.

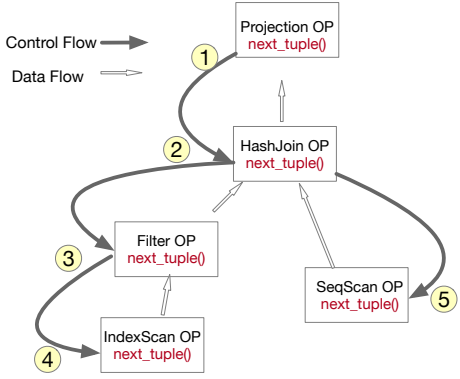
Iterator model example (1): SeqScan Operator

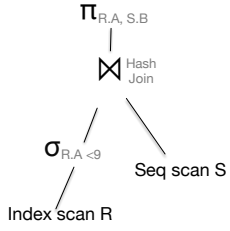
```
class SeqScanOperator : public Operator {
public:
  SeqScanOperator(Table *table) : table(table) {}
  Status init() override {
    iter = table->begin();
    return Status::InitOk;
  }
  Status next_tuple(Tuple &tuple) override {
    if (iter != table->end()) {
      tuple = iter.get_tuple();
  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





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

Announcement

- Lab2 has been released. DDL: 2025/11/15.



Lab2-聚合算子实现

最后更新 大约 19 小时前

DDL: 2025年11月15日

Lab2实验目的

在 Lab2 中我们将为 MiniOB 实现聚合算子，使系统能够正确执行如下查询语句：

```
-- CREATE TABLE t(int a, int b);
SELECT max(a) FROM t;                          -- Q1
SELECT max(a) FROM t GROUP BY b;               -- Q2
SELECT max(a) FROM t GROUP BY b HAVING count(a) > 2;  -- Q3
```

本次 **Lab2** 共包含两个任务

- **任务 1**：实现 SQL 中的基础聚合函数。完成后，MiniOB 将能够正确执行 **Q1**。
- **任务 2**：在任务 1 的基础上，完善带有 GROUP BY 的聚合逻辑，使系统能够支持 HAVING 子句。完成任务2后，MiniOB 可以正确执行 **Q1, Q2** 和 **Q3**。

- Please start early and pay attention to the tips provided by the TA.

Algorithms for Relational Operations

Notations

- 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

Sequential scan

- Scan table R sequentially and process the query
  - Selection over R
  - Projection of R without duplicate elimination
- I/O cost: $\mathrm{P}(\mathrm{R})$
- Not counting the cost of writing the result out
  - Maybe not needed – results may be pipelined into another operator
  - Same for the algorithms discussed later

Why sorting

- Query may require output be sorted.
  - E.g., `SELECT * from student ORDER BY credit DESC;`
- Several relational operators can be implemented efficiently with sorting.
  - E.g., duplication elimination, aggregation, merge join, set operations.
- External sorting is required when data cannot fit in memory.

External merge sorting

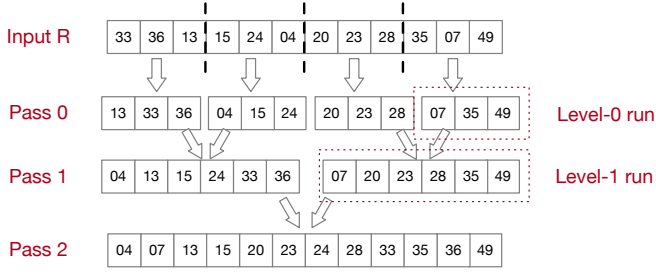
A divide-and-conquer approach to sort a large relation R that cannot fit in memory.

Recall that we have B pages available in the buffer pool.

- Pass 0: read B pages of R each time, sort them, and write out a level-0 run.
- Pass 1: merge $\mathrm{B}-1$ level-0 runs each time, and write out a level-1 run.
- Pass 2: merge $\mathrm{B}-1$ level-1 runs each time, and write out a level-2 run.
- ...
- Final pass produces one sorted run.

External merge example

- $B = 3$, i.e., 3 pages available in buffer pool.
- Each page holds only one tuple.
- In pass 0, all 3 pages are used for sorting.
- In pass i, where $i \geqslant 1$, $B-1=2$ pages are used for input, and 1 page for output.



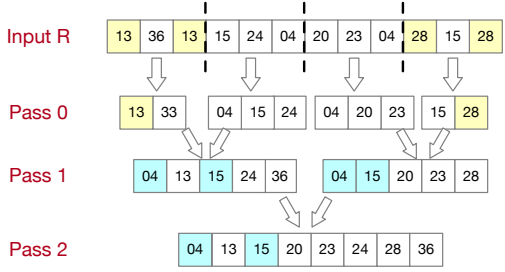
Cost analysis

#(Passes)	$\log_{B-1}\lceil P(R)/B\rceil + 1$
#(Read Pages)	$P(R) * (\log_{B-1}\lceil P(R)/B\rceil + 1)$
#(Write Pages)	$P(R) * \log_{B-1}\lceil P(R)/B\rceil$
Total cost	$2P(R)*\log_{B-1}\lceil P(R)/B\rceil + P(R)$

- Pass 0: read B pages of R each time, sort them, and write out a level-0 run.
- Pass i: merge $(B-1)$ level-$(i-1)$ runs each time, and write out a level-i run.
- Each pass read the entire relation and write it once.
- We do not include the output cost of the final pass as we have discussed.

Sort-based duplication elimination

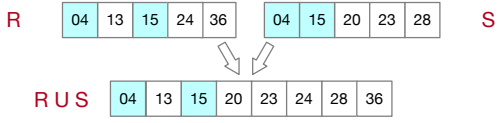
1. Perform external merge sort.
2. Eliminate duplicates during sort and merge.



Cost: same cost as sorting.

Sort-based set operations

- $R \cup S$, $R \cap S$, $R - S$ requires duplication elimination by default.
- Sort R and S in the same order.
- Scan the sorted R and S to produce the desired results and eliminate duplicates.
- Both R and S require only one pass of scan.
- Cost: sorting cost + $P(R) + P(S)$



Sort-based aggregation

- Sort the tuples on the GROUP BY attributes
- Perform a sequential scan over the sorted data to compute the aggregation.
  – This can be fused into the final pass of sorting.
- Apply partial aggregation on the fly.
- The output will be sorted on the attributes.
- Cost: same cost as sorting.

```
SELECT dept_name, AVG(salary)
FROM instructor
GROUP BY dept_name
```

Agg	Running value
MIN	min
MAX	max
COUNT	count
SUM	sum
AVG	(count, sum)

Naive nested loop join

```
1. for each tuple t_r in R do
2.     for each tuple t_s in S do
3.         if θ(t_r, t_s) then
4.             add t_r ⋈ t_s to the result
```

Figure: Algorithm for $R \bowtie_\theta S$

- The most basic join algorithm to compute join $R \bowtie_\theta S$.
- R: the outer table, S: the inner table.
- Require no indices and can be used with any kind of join conditions.

Naive nested loop join

```
1. for each tuple t_r in R do
2.     for each tuple t_s in S do
3.         if θ(t_r, t_s) then
4.             add t_r ⋈ t_s to the result
```

Figure: Algorithm for $R \bowtie_\theta S$

- The most basic join algorithm to compute join $R \bowtie_\theta S$.
- R: the outer table, S: the inner table.
- Require no indices and can be used with any kind of join conditions.
- Cost: $P(R) + |R| * P(S)$
- Buffer pool requirement: $B = 3$
  - Two buffer pool pages for input, and one for output.

Example

A	B
10	a
20	b
20	c
40	d

R

A	C
50	e
20	f
20	g
30	h
40	i
50	j

S

A	B	C
20	b	f
20	b	g
20	c	f
20	c	g
40	d	i

$R \bowtie S$

- $|R| = 4$, $|S| = 6$, $P(R) = 2$, $P(S) = 3$.
- If R is the outer table, then the cost is 14.
- If S is the outer table, then the cost is 15.

Blocked nested loop join

Idea: Process tables on a per-page basis, rather than on a per-tuple basis.

```
1. for each page P_r in R do
2.     for each page P_s in S do
3.         for each tuple t_r in P_r do
4.             for each tuple t_s in P_s do
5.                 if θ(t_r, t_s) then
6.                     add t_r ⋈ t_s to the result
```

Figure: Improved algorithm for $R \bowtie_\theta S$

Example

A	B
10	a
20	b
20	c
40	d

R

A	C
50	e
20	f
20	g
30	h
40	i
50	j

S

A	B	C
20	b	f
20	b	g
20	c	f
20	c	g
40	d	i

$R \bowtie S$

- $|R| = 4$, $|S| = 6$
- $P(R) = 2$, $P(S) = 3$.
- If R is the outer table, then the cost is 8.
- If S is the outer table, then the cost is 9.

Cost analysis

- Cost: $P(R) + P(R) * P(S)$
- Buffer pool requirement: $B = 3$

Optimization: If B pages are available in the buffer pool for the join operation, then

- $B - 2$ pages for scanning the outer table R
- One page for inner table scan
- The rest page for buffering the output
- Total cost: $P(R) + \lceil P(R)/(B-2) \rceil * P(S)$

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 ⋈ t_s to the result
```

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

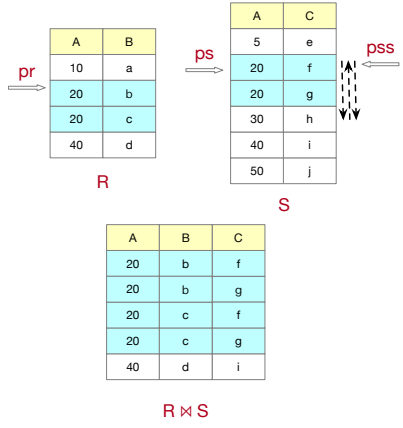
- Cost analysis: $P(R) + |R| * C$.
- C is the I/O cost of an index lookup, which is $2 \sim 4$ I/O's typically.
- If both R and S support index lookup, better pick the smaller one as the outer relation.

Merge join

- Require equality predicate, e.g., equi-joins or natural joins.
- If R or S is not sorted by the join attributes, then sort it first.
- All tuples with the same value on the joined attributes are in consecutive order.
- Merge scan the sorted tables and emit tuples that match.

Merge join

```
1.  /* ps/pr points to the first tuple of R/S */
2.  while pr! = EOF & ps! = EOF do
3.    while t_pr[A] < t_ps[A] do ++pr;
4.    while t_pr[A] > t_ps[A] do ++ps;
5.    while t_pr[A] = t_ps[A] do
6.      pss := ps; /* set pss to the first match */
7.      while t_pr[A] = t_pss[A] do
8.        add t_pr ⋈ t_pss to result;
9.        ++pss;
10.     ++pr;
11.   ps := pss; /* all matches emitted, advance ps */
```



Cost analysis

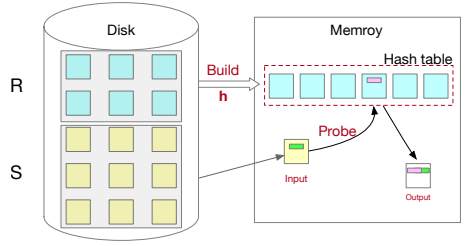
- Most cases: Sorting + $P(R) + P(S)$.
- Assumption: Every set of match candidates in S can fit in buffer pool.

- Worst case: Sorting + $P(R) + P(R) * P(S)$
- Assumption: Everything joins and $B = 3$.

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

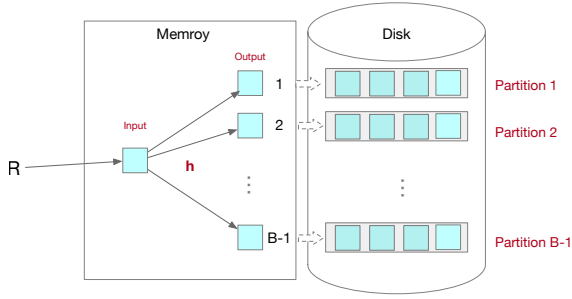
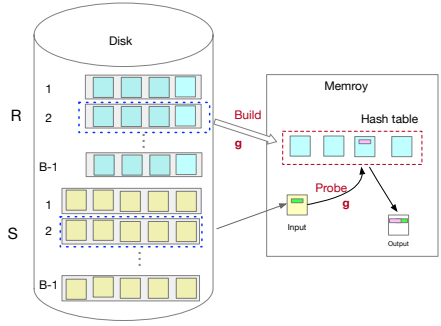


Figure: Partition $\mathtt{R}$ with $\mathtt{h}$ (need to do the same for $\mathtt{S}$)

- Partition both $\mathtt{R}$ into $\mathtt{B}-1$ partitions, using a hash function $\mathtt{h}$ on the join attributes.
- A buffer block/page is reserved as the output buffer for each partition.
- Partition table $\mathtt{S}$ 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.
- 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 is too large for memory?

Hash-based algorithms

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

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 $R \bowtie S$