資料庫管理 HW05

B12508026 戴偉璿

December 1, 2025

1. If we read the outer loop into $B-2$ blocks, the total cost is M(number of outer blocks)+$\lceil \frac{M}{B-2} \rceil \times N$(number of inner loop read times). Thus, the total cost is $\lceil \frac{M}{B-2} \rceil \times N + M$ I/Os. In the other way, if we read the inner loop into $B-2$ blocks, the total cost is $N + \lceil \frac{N}{B-2} \rceil \times M$ I/Os. Therefore, the total cost is $\min(\lceil \frac{M}{B-2} \rceil \times N + M, N + \lceil \frac{N}{B-2} \rceil \times M)$ I/Os. Thus, these two methods are roughly equal. However, in practical, we usually put smaller relation in the outer loop to reduce the number of passes of the inner loop. In this case, place the smaller relation into $B-2$ blocks is better.

2.

   (a) I don't think this plan is more efficient. There are no join keys between `PRODUCT` and `SALES`. If we join these two relations first, it would cause cartesian product, which would produce a very large intermediate relation.

   (b) To achieve projection pushdown, I project the needed attributes at the first step. Next step would be handle the predicate pushdown. I apply the selection `s.store_id=1` and `sd,unit_price>=20` at this phase. Now I need to decide which join should be performed first. I have two joins: `PRODUCT` $\bowtie$ `SALES_DETAIL` and `SALES` $\bowtie$ `SALES_DETAIL`. Any way, `SALES_DETAIL` must be joined first. Thus, I need to choose between `PRODUCT` $\bowtie$ `SALES_DETAIL` and `SALES` $\bowtie$ `SALES_DETAIL`. Finally I decide to join `SALES` and `SALES_DETAIL` first because there is a selection `s.store_id=1` on `SALES`, which would reduce the number of tuples in the intermediate relation. Besides, the join key `receipt_no` may be more selective than `product_id`. The overall plan is shown as Fig .1.

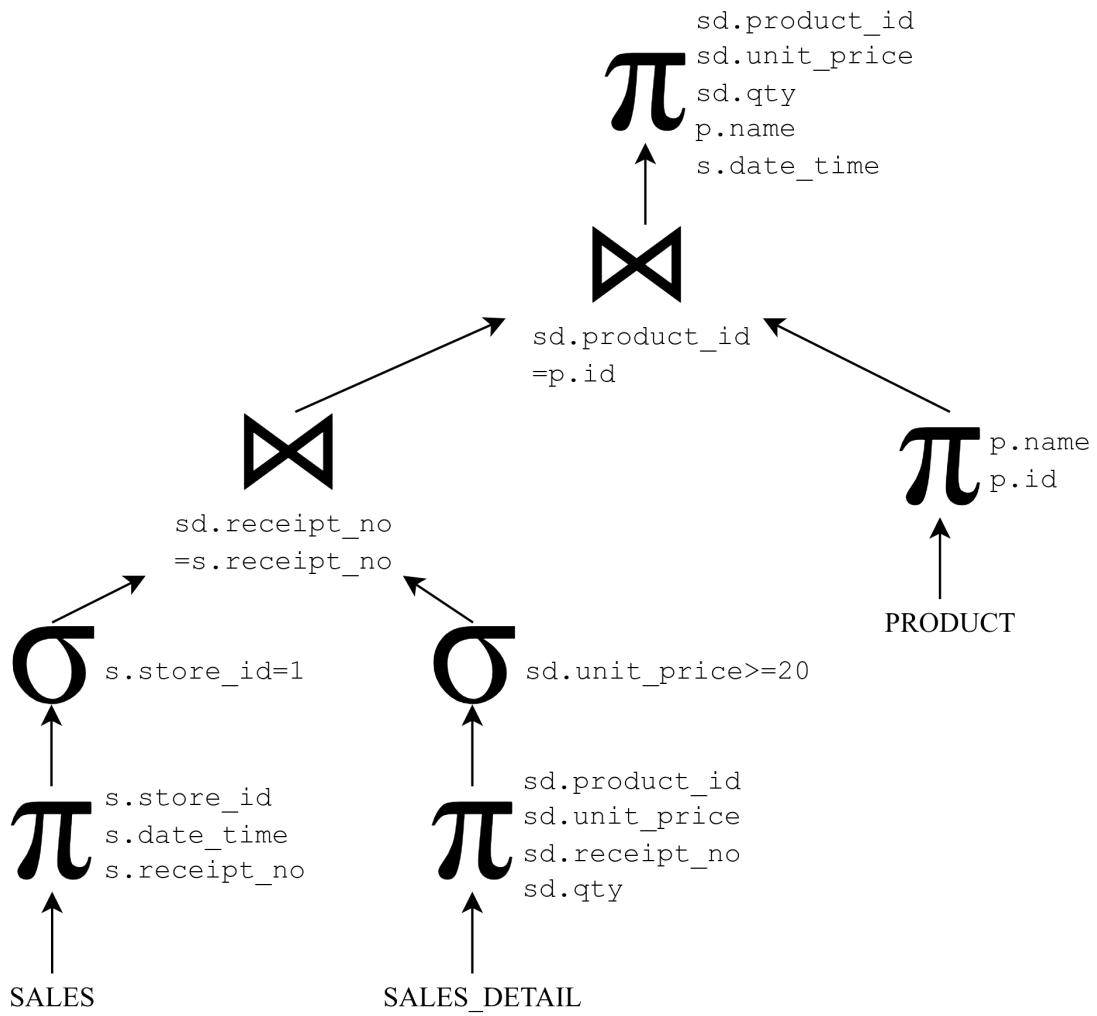

Figure 1: Query Plan with Projection and Predicate Pushdown

(c) The query plan generated by postgreSQL is shown as Fig .2.

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=21.71..24.47 rows=1 width=138)
   ->  Hash Join  (cost=21.56..23.61 rows=1 width=20)
         Hash Cond: ((sd.receipt_no)::text = (s.receipt_no)::text)
         ->  Seq Scan on sales_detail sd  (cost=0.00..1.86 rows=69 width=23)
               Filter: (unit_price >= 20)
         ->  Hash  (cost=21.50..21.50 rows=5 width=56)
               ->  Seq Scan on sales s  (cost=0.00..21.50 rows=5 width=56)
                     Filter: (store_id = 1)
   ->  Index Scan using product_pkey on product p  (cost=0.15..0.86 rows=1 width=122)
         Index Cond: (id = sd.product_id)
(10 rows)
```

Figure 2: Query Plan Generated by PostgreSQL

The DBMS first pushes down the filters so that the tables become smaller before

any joins. Then it estimates how many rows remain after each step and chooses the join order that keeps the intermediate results as small as possible. This is why it joins `SALES_DETAIL` and `SALES` first using a Hash Join, and only later joins `PRODUCT` using an index.

(d) The query plan of this query is same as Fig .2. These two SQL statements produce the same query plan because PostgreSQL rewrites the subquery, pushes down the predicates, and performs cost-based join reordering. The DBMS does not execute the subquery independently; instead, it merges it into the global logical plan and chooses the most efficient join order based on cardinality estimation.

3. (a) The algorithm is shown as follow:

```cpp
#include<bits/stdc++.h>
using namespace std;

struct st1 {
    string sid, cid, name;
};
struct st2 {
    string cid, name;
};

inline int hash_func(const string &s) {
    int result=0;
    // calculate hash value
    for(char c:s)
        result=(result+(int)c)%500;
    return result;
}

int main() {
    ifstream f1("reserved_ticket.csv"), f2("name.csv");
    string line, s1, s2;// buffers for parsing
    // utilize vector to store multiple records in same hash bucket
    vector<st1> v1[500];// reserved tickets
    vector<st2> v2[500];// name
    while(getline(f1, line)){
        // parse CSV line
        stringstream ss(line);
        getline(ss, s1, ',');// sid
        getline(ss, s2, ',');// cid
        // insert record into corresponding hash bucket
        v1[hash_func(s2)].push_back({s1, s2, ""});
    }
    while(getline(f2, line)){
        stringstream ss(line);
        getline(ss, s1, ',');// cid
        getline(ss, s2, ',');// name

        int hashed_result=hash_func(s1);
        if(v1[hashed_result].size()){
            for(auto &rec : v1[hashed_result]){
                if(rec.cid==s1){
                    rec.name=s2;
```

```
                }
            }
        }
    }
    cout<<"start\n";
    int rows=0;
    for(int i=0;i<500;i++){
        if(v1[i].size()){
            for(const auto &rec : v1[i]) {
                if(rec.name!="")
                    cout<<rec.sid<<","<<rec.name<<"\n", rows++;
            }
        }
    }
    cout<<"total rows: "<<rows<<"\n";

    return 0;
}
```

And the number of records in the first 100 hash buckets are shown as Table .1.

id	#	id	#	id	#	id	#	id	#	id	#	id	#	id	#	id	#	id	#
0	0	10	0	20	6	30	253	40	887	50	674	60	291	70	27	80	0	90	0
1	0	11	0	21	3	31	349	41	862	51	646	61	240	71	9	81	0	91	0
2	0	12	0	22	11	32	397	42	899	52	583	62	219	72	9	82	0	92	0
3	0	13	0	23	29	33	479	43	913	53	553	63	200	73	11	83	0	93	0
4	0	14	0	24	41	34	545	44	862	54	512	64	145	74	6	84	0	94	0
5	0	15	0	25	74	35	634	45	856	55	433	65	112	75	1	85	0	95	0
6	0	16	0	26	63	36	657	46	824	56	434	66	91	76	0	86	0	96	0
7	0	17	0	27	98	37	782	47	783	57	385	67	75	77	0	87	0	97	0
8	0	18	0	28	172	38	835	48	837	58	384	68	40	78	0	88	0	98	0
9	0	19	1	29	193	39	814	49	716	59	308	69	27	79	0	89	0	99	0

Table 1: Number(#) of Records in the First 100 Hash Buckets

(b) The algorithm is shown as follow:

```
#include<bits/stdc++.h>
using namespace std;

struct st1 {
    string sid, cid, name;
};
struct st2 {
    string cid, name;
};

int main() {
    ifstream f1("reserved_ticket.csv"), f2("name.csv");
    string line, s1, s2;// buffers for parsing
```

```cpp
    // utilize vector to store multiple records in same hash bucket
    vector<st1> v1;// reserved tickets
    vector<st2> v2;// name
    while(getline(f1, line)){
        stringstream ss(line);
        getline(ss, s1, ',');
        getline(ss, s2, ',');
        v1.push_back({s1, s2, ""});
    }
    while(getline(f2, line)){
        stringstream ss(line);
        getline(ss, s1, ',');
        getline(ss, s2, ',');
        v2.push_back({s1, s2});
    }
    sort(v1.begin(), v1.end(), [](const st1 &a, const st1 &b){
        return a.cid<b.cid;
    });
    sort(v2.begin(), v2.end(), [](const st2 &a, const st2 &b){
        return a.cid<b.cid;
    });

    int pt1=0, pt2=0;
    while(pt1<v1.size() && pt2<v2.size()){
        if(v1[pt1].cid==v2[pt2].cid){
            v1[pt1].name=v2[pt2].name;
            pt1++;// someone may have multiple tickets, thus only move pt1
        }else if(v1[pt1].cid<v2[pt2].cid){
            pt1++;
        }else{
            pt2++;
        }
    }
    cout<<"start\n";
    int rows=0;
    for(const auto &rec : v1){
        if(rec.name!="")
            cout<<rec.sid<<","<<rec.name<<"\n", rows++;
    }
    cout<<"total rows: "<<rows<<"\n";

    return 0;
}
```

(c) After running both codes, I found that hash join costs 124 ms while sort-merge join costs 14 ms. Sort-merge join is faster than hash join in this case. The bottleneck of hash join is the hash function. The algorithm must scan the entire string of `cid` to compute the hash value and later perform a full string comparison inside the hash bucket. Besides, I observe that the hashed results concentrate unevenly in some buckets, which may cause lots of collisions, and this is harmful to any hashing algorithm.

On the other hand, sort-merge join only needs to compare the strings of `cid` until it can determine which one is larger. The bottleneck of sort-merge join is the sorting

operation, but the `std::sort()` function is one of the most optimized functions in C++ STL, and the elements are relativelly small. Thus, the overall performance of sort-merge join is better than hash join in this case.

Overall, the performance of hash join highly depends on the quality of the hash function and the distribution of the hashed results. If there are many collisions, the performance would degrade significantly. Though it has a better expected time complexity of O(n), its practical performance may be worse than other algorithms with higher time complexity in some cases. If the amount of data is small enough to fit into memory, sort-merge join would be a better choice due to its lower constant factors (but the difference may be negligible for very small data).

4. ACID refers to Atomicity, Consistency, Isolation, and Durability, they guarantee that database transactions are processed reliably. BASE refers to Basically Available, Soft state, Eventual consistency, which is an alternative to the ACID model for database systems that prioritize availability over strict consistency. In short, ACID focuses on ensuring data integrity and consistency tasks such as bank account transactions, while BASE emphasizes system availability and scalability tasks such as social media platforms.

5. (a) It is reasonable to use NoSQL databases for this application. From the perspective of ACID versus BASE, the storage of receipt records does not require strong consistency. Each receipt is written once and rarely modified, and there is no need for immediate synchronization across all replicas. Eventual consistency is sufficient because temporary inconsistency does not harm correctness at the application level. Adopting a BASE model allows the system to tolerate replication delays, improving availability and write throughput—both crucial for high-traffic services that continuously ingest large volumes of receipts.

   In addition, NoSQL systems are designed for horizontal scalability and can efficiently manage massive datasets. Receipt data often includes semi-structured or nested fields (such as item lists, merchant metadata, or variable formats from different issuers), which align well with schema-flexible document or column-family stores. For applications that primarily perform large-scale aggregation, analytics, and high-frequency writes, NoSQL architectures provide more suitable performance and scalability than traditional ACID-oriented relational databases.

   (b) Sharding the database by region is a reasonable strategy. Consumer shopping patterns are typically geographically localized, meaning that most receipts, merchants, and related queries originate within the same area. By partitioning data based on region, the system can keep the majority of read and write operations on a single shard. This reduces cross-shard communication, lowers coordination overhead, and improves both latency and throughput.

(c) I would shard the user table (especially the student role) by timestamp.

Each year, the number of newly enrolled students is roughly similar, so partitioning by enrollment year naturally keeps each shard at a comparable size. This avoids creating a hotspot shard and helps distribute the load evenly.

In addition, the system includes a feature that hard-deletes users one year after soft deletion, meaning older records become cold data. Timestamp-based sharding allows these old shards to be archived or removed easily, making data management more efficient.

6. (a) Transfer `p.id`, `p.name` from Place 1 to Place 2, it costs $100 \times (4+50) = 5400$ Bytes. Transfer `sd.receipt_no`, `sd.product_id`, `sd.unit_price`, `sd.qty` from Place 3 to Place 2, it costs $5000 \times (10+4+4+4) = 110000$ Bytes. Total network cost is 115400 Bytes.

(b) Transfer `p.id`, `p.name` from Place 1 to Place 3, it costs $100 \times (4+50) = 5400$ Bytes. Transfer `s.receipt_no` from Place 2 to Place 3, it costs $400 \times 10 = 4000$ Bytes. (There are 400 sales occured at `store_id=1`) Finally, transfer `sd.receipt_no`, `sd.product_id`, `sd.unit_price`, `sd.qty`, `p.name` from Place 3 to Place 2, it costs $1000 \times (10+4+4+4+50) = 360000$ Bytes. Total network cost is 375400 Bytes.