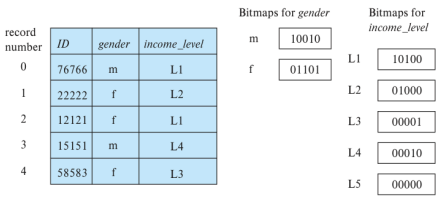
INDICES

Bitmap

Each bit in a bitmap corresponds to a possible item or condition, with a bit set to 1 indicating presence or true, and a bit set to 0 indicating absence or false.



record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Figure 14.28 Bitmap indices on relation *instructor_info*.

B+ tree

B+ tree is a type of self-balancing tree data structure that maintains data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. It is an extension of the B-tree and is extensively used in databases and filesystems for indexing. B+ tree is **Balanced**; Order (n): Defined such that each node (except root) can have at most n children (pointers) and at least $\lceil \frac{n}{2} \rceil$ children; **Internal nodes hold** between $\lceil n/2 \rceil - 1 \lceil n/2 \rceil - 1$ and $n-1 n-1$ keys; Leaf nodes also hold between $\lceil \frac{n}{2} \rceil - 1$ and $n-1$ keys but also store all data values corresponding to the keys; **Leaf Nodes Linked**: Leaf nodes are linked together, making range queries and sequential access very efficient.

- **Insert (key, data)**:
  - Insert key in the appropriate leaf node in sorted order;
  - If the node overflows (more than $n-1$ keys), split it, add the middle key to the parent, and adjust pointers;
    1. Leaf split: 1 to $\lceil \frac{n}{2} \rceil$ and $\lceil \frac{n}{2} \rceil + 1$ to n as two leafs. Promote the lowest from the 2nd one.
    2. Node split: 1 to $\lceil \frac{n+1}{2} \rceil - 1$ and $\lceil \frac{n}{2} \rceil + 1$. $\lceil \frac{n+1}{2} \rceil - 1$ gets moved up.
  - If a split propagates to the root and causes the root to overflow, split the root and create a new root. Note: root can contain less than $\lceil \frac{n}{2} \rceil - 1$ keys.
- **Delete (key)**:
  - Remove the key from the leaf node.
  - If the node underflows (fewer than $\lceil \frac{n}{2} \rceil - 1$ keys), keys and pointers are redistributed or nodes are merged to maintain minimum occupancy. -

  Adjustments may propagate up to ensure all properties are maintained.

Hash-index

Hash indices are a type of database index that uses a hash function to compute the location (hash value) of data items for quick retrieval. They are particularly efficient for equality searches that match exact values.

Hash Function: A hash function takes a key (a data item's attribute used for indexing) and converts it into a hash value. This hash value determines the position in the hash table where the corresponding record's pointer is stored. **Hash Table**: The hash table stores pointers to the actual data records in the database. Each entry in the hash table corresponds to a potential hash value generated by the hash function.

ALGORITHMS

Nested-loop join

Nested Loop Join: A nested loop join is a database join operation where each tuple of the outer table is compared against every tuple of the inner table to find all pairs of tuples which satisfy the join condition. This method is simple but can be inefficient for large datasets due to its high computational cost.

```
Simplified version (to get the idea)
for each tuple tr in r: (for each tuple ts in s: test pair (tr, ts))
```

Block transfer cost: $n_r * b_s + b_r$ block transfers would be required, where b_r – blocks in relation$r$, same for s.

Block-nested join

Block Nested Loop Join: A block nested loop join is an optimized version of the nested loop join that reads and holds a block of rows from the outer table in memory and then loops through the inner table, reducing the number of disk accesses and improving performance over a standard nested loop join, especially when indices are not available.

```
Simplified version (to get the idea)
for each block Br of r: for each block Bs of s:
  for each tuple tr in r: (for each tuple ts in s: test pair (tr, ts))
```

Block transfer cost: $b_r * b_s + b_r$, b_r – blocks in relation r, same for s.

Merge join

Merge Join: A merge join is a database join operation where both the outer and inner tables are first sorted on the join key, and then merged together by sequentially scanning through both tables to find matching pairs. This method is highly efficient when the tables are **already sorted** or can be **sorted quickly**, minimizes random disk access. Merge-join method is efficient; the number of block transfers is equal to the sum of the number of blocks in both files, $b_r + b_s$. Assuming that bb buffer blocks are allocated to each relation, the number of disk seeks required would be $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ disk seeks

1. Sort Both Tables: If not already sorted, the outer table and the inner table are sorted based on the join keys.
2. Merge: Once both tables are sorted, the algorithm performs a merging operation similar to that used in merge sort:
   1. Begin with the first record of each table.
   2. Compare the join keys of the current records from both tables.
      1. If the keys match, join the records and move to the next record in both tables.
      2. If the join key of the outer table is smaller, move to the next record in the outer table.
      3. If the join key of the inner table is smaller, move to the next record in the inner table.
   3. Continue this process until all records in either table have been examined.
3. Output the Joined Rows;

Hash-join

Hash Join: A hash join is a database join operation that builds an in-memory hash table using the join key from the smaller, often called the build table, and then probes this hash table using the join key from the larger, or probe table, to find matching pairs. This technique is very efficient for **large datasets** where **indexes are not present**, as it reduces the need for nested loops.

- h is a hash function mapping JoinAttrs values to $\{0, 1, ..., n_h\}$, where JoinAttrs denotes the common attributes of r and s used in the natural join.
- $r_0, r_1, ...$, rnh denote partitions of r tuples, each initially empty. Each tuple $t_r \in r$ is put in partition r_i, where $i = h(t_r[\text{JoinAttrs}])$.
- $s_0, s_1, ..., s_{n_h}$ denote partitions of s tuples, each initially empty. Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[\text{JoinAttrs}])$.

Cost of block transfers: $3(b_r + b_s) + 4n_h$. The hash join thus requires $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2n_h$ seeks.

b_b blocks are allocated for the input buffer and each output buffer.

1. Build Phase:
   1. Choose the smaller table (to minimize memory usage) as the "build table."
   2. Create an in-memory hash table. For each record in the build table, compute a hash on the join key and insert the record into the hash table using this hash value as an index.
2. Probe Phase:
   1. Take each record from the larger table, which is often referred to as the "probe table."
   2. Compute the hash on the join key (same hash function used in the build phase).
   3. Use this hash value to look up in the hash table built from the smaller table.
   4. If the bucket (determined by the hash) contains any entries, check each entry to see if the join key actually matches the join key of the record from the probe table (since hash functions can lead to collisions).
3. Output the Joined Rows.

RELATIONAL-ALGEBRA

Equivalence rules

1. $\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right)$
2. $\sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right) \equiv \sigma_{\theta_2}\left(\sigma_{\theta_1}(E)\right)$
3. $\Pi_{L_1}\left(\Pi_{L_2}\left(...\left(\Pi_{L_n}(E)\right)...\right)\right) \equiv \Pi_{L_1}(E)$ – only the last one matters.
4. Selections can be combined with Cartesian products and theta joins: $\sigma_{\theta(E_1 \times E_2)} \equiv E_1 \underset{\theta}{\bowtie} E_2$ - This expression is just the definition of the theta join |||| $\sigma_{\theta_1}\left(E_1 \underset{\theta_2}{\bowtie} E_2\right) \equiv E_1 \underset{\theta_1}{\bowtie} \wedge \theta_2 E_2$

5. $E_1 \underset{\theta}{\bowtie} E_2 \equiv E_2 \underset{\theta}{\bowtie} E_1$
6. Join associativity: $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$ |||| $\left(E_1 \bowtie_{\theta_1} E_2\right) \bowtie_{\theta_2 \wedge \theta_3}$ ||| $E_3 \equiv E_1 \bowtie_{\theta_1 \vee \theta_3} \left(E_2 \bowtie_{\theta_2} E_3\right)$
7. Selection distribution: $\sigma_{\theta_1}\left(E_1 \underset{\theta}{\bowtie} E_2\right) \equiv \left(\sigma_{\theta_1}(E_1)\right) \underset{\theta}{\bowtie} E_2$; $\sigma_{\theta_1 \wedge \theta_2}\left(E_1 \underset{\theta}{\bowtie} E_2\right) \equiv \left(\sigma_{\theta_1}(E_1)\right) \underset{\theta}{\bowtie} \left(\sigma_{\theta_2}(E_2)\right)$
8. Projection distribution: - $\Pi_{L_1 \cup L_2}\left(E_1 \underset{\theta}{\bowtie} E_2\right) \equiv \left(\Pi_{L_1(E_1)}\right) \underset{\theta}{\bowtie} \left(\Pi_{L_2}(E_2)\right)$ |||| $\Pi(L_1 \cup L_2)\left(E_1 \underset{\theta}{\bowtie} E_2\right) \equiv \Pi_{L_1 \cup L_2}\left(\left(\Pi_{L_1 \cup L_3}(E_1)\right) \underset{\theta}{\bowtie} \left(\Pi_{L_2 \cup L_4}(E_2)\right)\right)$
9. Union and intersection commmutativity: E1 ∪E2 =E2 ∪E1 |||| - E1 ∩E2 =E2 ∩E1
10. Set union and intersection are associative: (E1 ∪E2) ∪E3 =E1 ∪(E2 ∪E3) |||| (E1 ∩E2) ∩E3 =E1 ∩(E2 ∩E3);
11. The selection operation distributes over the union, intersection, and set-difference operations: σθ(E1 ∪E2) =σθ(E1) ∪σθ(E2) |||| σθ(E1 ∩E2) =σθ(E1) ∩σθ(E2) |||| σθ(E1 −E2) =σθ(E1) −σθ(E2) |||| σθ(E1 ∩E2) =σθ(E1) ∩E2 |||| σθ(E1 −E2) =σθ(E1) −E2 |||| 12.
12. The projection operation distributes over the union operation - ΠL(E1 ∪E2) =(ΠL(E1)) ∪(ΠL(E2))

CONCURRENCY

Conflict

We say that I and J conflict if they are operations by **different transactions** on the **same data item**, and at least one of these instructions is a **write** operation. For example: I = read(Q), J = read(Q) – Not a conflict; I = read(Q), J = write(Q) – Conflict; I = write(Q), J = read(Q) – Conflict; I = write(Q), J = write(Q) – Conflict.

Conflict-serializability

If a schedule S can be transformed into a schedule S' by a series of swaps of non- conflicting instructions, we say that S and S' are **conflict equivalent**. We can swap only *adjacent* operations.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule.

Serializability graph

Simple and efficient method for determining the conflict seriazability of a schedule. Consider a schedule S. We construct a directed graph, called a precedence graph, from S. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

1. T_i executes `write(Q)` before T_j executes `read(Q)`.
2. T_i executes `read(Q)` before T_j executes `write(Q)`.
3. T_i executes `write(Q)` before T_j executes `write(Q)`.

If the precedence graph for S has a cycle, then schedule S is not conflict serializable. If the graph contains no cycles, then the schedule S is conflict serializable.

Standard isolation levels

- **Serializable** usually ensures serializable execution.
- **Repeatable** read allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it. However, the transaction may not be serializable
- **Read committed** allows only committed data to be read, but does not require re- peatable reads.
- **Read uncommitted** allows uncommitted data to be read. Lowest isolation level allowed by SQL.

Protocols

We say that a schedule S is **legal** under a given locking protocol if S is a possible schedule for a set of transactions that follows the rules of the locking protocol. We say that a locking protocol ensures conflict serializability if and only if all legal schedules are **conflict serializable**; in other words, for all legal schedules the associated ⟶relation is acyclic.

Lock-based

2-phased lock protocol

The Two-Phase Locking (2PL) Protocol is a concurrency control method used in database systems to ensure serializability of transactions. The protocol involves two distinct phases: **Locking Phase (Growing Phase):** A transaction may acquire locks but cannot release any locks. During this phase, the transaction continues to lock all the resources (data items) it needs to execute.
Unlocking Phase (Shrinking Phase): The transaction releases locks and cannot acquire any new ones. Once a transaction starts releasing locks, it moves into this phase until all locks are released.

Problems of locks

Deadlock is a condition where two or more tasks are each waiting for the other to release a resource, or more than two tasks are waiting for resources in a circular chain.
Starvation (also known as indefinite blocking) occurs when a process or thread is perpetually denied necessary resources to process its work. Unlike deadlock, where everything halts, starvation only affects some while others progress.

Timestamp-based

Timestamp Assignment: Each transaction is given a unique timestamp when it starts. This timestamp determines the transaction's temporal order relative to others. **Read Rule:** A transaction can read an object if the last write occurred by a transaction with an earlier or the same timestamp. **Write Rule:** A transaction can write to an object if the last read and the last write occurred by transactions with earlier or the same timestamps.

Validation-based

Assumes that conflicts are rare and checks for them only at the end of a transaction. **Working Phase:** Transactions execute without acquiring locks, recording all data reads and writes. **Validation Phase:** Before committing, each transaction must validate that no other transactions have modified the data it accessed. **Commit Phase:** If the validation is successful, the transaction commits and applies its changes. If not, it rolls back and may be restarted.

Version isolation

LOGS

WAL principle

Write Ahead Logging – Any change to data (update, delete, insert) must be recorded in the log before the actual data is written to the disk. This ensures that if the system crashes before the data pages are saved, the changes can still be reconstructed from the log records during recovery.

Recovery algorithm

In the **redo phase**, the system replays updates of all transactions by scanning the log forward from the last checkpoint. The specific steps taken while scanning the log are as follows:

1. The list of transactions to be rolled back, undo-list, is initially set to the list L in the $<$ checkpoint $L>$ log record.
2. Whenever a normal log record of the form $< T_i, X_j, V_1, V_2 >$, or a redo- only log record of the form $< T_i, X_j, V_2 >$ is encountered, the operation is redone; that is, the value V_2 is written to data item X_j.
3. Whenever a log record of the form $< T_i$ start $>$ is found, T_i is added to undo-list.
4. Whenever a log record of the form $< T_i$ abort $>$ or $< T_i$ commit $>$ is found, T_i is removed from undo-list.

At the end of the redo phase, undo-list contains the list of all transactions that are incomplete, that is, they neither committed nor completed rollback before the crash.
In the **undo phase**, the system rolls back all transactions in the undo-list. It performs rollback by scanning the log backward from the end:

1. Whenever it finds a log record belonging to a transaction in the undo-list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.
2. When the system finds a $< T_i$ start $>$ log record for a transaction T_i in undo- list, it writes a $< T_i$ abort $>$ log record to the log and removes T_i from undo- list.
3. The undo phase terminates once undo-list becomes empty, that is, the system has found $< T_i$ start $>$ log records for all transactions that were initially in undo-list.

Log types

- $< T_i, X_j, V_1, V_2 >$ – an update log record, indicating that transaction T_i has performed a write on data item X_j. X_j had value V_1 before the write and has value V_2 after the write.
- $< T_i$ start $>$ – T_i has started.
- $< T_i$ commit $>$ – T_i has committed.
- $< T_i$ abort $>$ – T_i has aborted.