Lecture #01: Relational Model & Algebra

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Databases

A *database* is an organized collection of inter-related data that models some aspect of the real-world (e.g., modeling the students in a class or a digital music store). People often confuse "databases" with "database management systems" (e.g., MySQL, Oracle, MongoDB, Snowflake). A database management system (DBMS) is the software that manages a database.

Consider a database that models a digital music store (e.g., Spotify). Let the database hold information about the artists and which albums those artists have released.

2 Flat File Strawman

Database is stored as comma-separated value (CSV) files that the DBMS manages. Each entity will be stored in its own file. The application has to parse files each time it wants to read or update records.

Keeping along with the digital music store example, there would be two files: one for artist and the other for album.

Each entity has its own set of attributes, so in each file, different records are delimited by new lines, while each of the corresponding attributes within a record are delimited by a comma.

Example: An artist could have a name, year, and country attributes, while an album has name, artist and year attributes.

The following is an example CSV file for information about artists with the schema (name, year, country):

```
"Wu-Tang Clan", 1992, "USA"
"Notorious BIG", 1992, "USA"
"GZA", 1990, "USA"
```

Issues with Flat File (for the digital music store example)

- **Data Integrity** How do we ensure that the artisi is the same for each album entry?What if somebody overwrites the album year with an invalid string? What if there are multiple artists on an album? What happens if we delete an artist that has albums?
- **Implementation** How do you find a particular record? What if we now want to create a new application that uses the same database? What if that application is running on a different machine? What if two threads try to write to the same file at the same time?
- **Durability** What if the machine crashes while our program is updating a record? What if we want to replicate the database on multiple machines for high availability?

issues with flat-file db storage :
- data integrity - implementation - durability

3 Database Management System

A *DBMS* is a software that allows applications to store and analyze information in a database.

A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases in accordance with some data model.

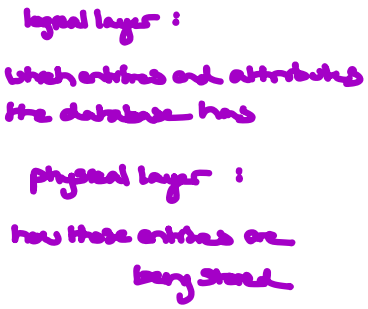
A *data model* is a collection of concepts for describing the data in database.

Examples: relational (most common), NoSQL (key/value, document, graph), array/matrix/vectors (for machine learning)

A *schema* is a description of a particular collection of data based on a data model.

Common Data Models

- Relational (Most DBMSs)
- Key/Value (NoSQL)
- Graph (NoSQL)
- Document/XML/Object (NoSQL)
- Wide-Column/Column-family (NoSQL)
- Array/Matrix/Vectors (Machine Learning)
- Hierarchical (Obsolete/Legacy/Rare)
- Network (Obsolete/Legacy/Rare)
- Multi-Value (Obsolete/Legacy/Rare)



Early DBMSs

Early database applications were difficult to build and maintain because there was a tight coupling between logical and physical layers.

The logical layer describes which entities and attributes the database has while the physical layer is how those entities and attributes are being stored. Early on, the physical layer was defined in the application code, so if we wanted to change the physical layer the application was using, we would have to change all of the code to match the new physical layer.

4 Relational Model

Ted Codd noticed that people were rewriting DBMSs every time they wanted to change the physical layer, so in 1969 he proposed the relational model to avoid this.

The relational model defines a database abstraction based on relations to avoid maintenance overhead. It has three key points:

- Store database in simple data structures (relations).
- Access data through high-level language, DBMS figures out best execution strategy.
- Physical storage left up to the DBMS implementation.

The relational data model defines three concepts:

- **Structure:** The definition of relations and their contents. This is the attributes the relations have and the values that those attributes can hold.
- **Integrity:** Ensure the database's contents satisfy constraints. An example constraint would be that any value for the year attribute has to be a number.
- **Manipulation:** How to access and modify a database's contents.

A *relation* is an unordered set that contains the relationship of attributes that represent entities. Since the relationships are unordered, the DBMS can store them in any way it wants, allowing for optimization. It is possible to have repeated elements in a relation.

A *tuple* is a set of attribute values (also known as its *domain*) in the relation. Originally, values had to be atomic or scalar, but now values can also be lists or nested data structures. Every attribute can be a special value, NULL, which means for a given tuple the attribute is undefined.

A relation with n attributes is called an n*-ary relation*. We will interchangeably use *relation* and *table* in this course. An n-ary relation is equivalent to a table with n columns.

Keys

A relation's *primary key* uniquely identifies a single tuple. Some DBMSs automatically create an internal primary key if you do not define one. A lot of DBMSs have support for autogenerated keys so an application does not have to manually increment the keys, but a primary key is still required for some DBMSs.

A *foreign key* specifies that an attribute from one relation has to map to a tuple in another relation. For example, we can include artist id (foreign key referring to the artist table) in the album table.

Constraints

A *constraint* is a user-defined condition that must hold for any instance of the database.

5 Data Manipulation Languages (DMLs)

Methods to store and retrieve information from a database. There are two classes of languages for this:

- **Procedural:** The query specifies the (high-level) strategy the DBMS should use to find the desired result based on sets / bags. For example, use a for loop to scan all records and count how many records are there to retrieve the number of records in the table.
- **Non-Procedural (Declarative):** The query specifies only *what* data is wanted and not *how* to find it. For example, use SQL `select count(*) from artist` to count how many records are there in the table.

6 Relational Algebra

Relational Algebra is a set of fundamental operations to retrieve and manipulate tuples in a relation. Each operator takes in one or more relations as inputs, and outputs a new relation. To write queries we can "chain" these operators together to create more complex operations.

Select

Select takes in a relation and outputs a subset of the tuples from that relation that satisfy a selection predicate. The predicate acts like a filter, and we can combine multiple predicates using conjunctions and disjunctions.

Syntax: $\sigma_{\text{predicate}}(R)$.

Example: $\sigma_{\text{a\_id='a2'}}(R)$

SQL: `SELECT * FROM R WHERE a_id = 'a2'`

Projection

Projection takes in a relation and outputs a relation with tuples that contain only specified attributes. You can rearrange the ordering of the attributes in the input relation as well as manipulate the values.

Syntax: $\pi_{A1,A2,\ldots,An}(R)$.

Example: $\pi_{b\_id\text{-}100,\ a\_id}(\sigma_{a\_id='a2'}(R))$

SQL: `SELECT b_id-100, a_id FROM R WHERE a_id = 'a2'`

Union

Union takes in two relations and outputs a relation that contains all tuples that appear in at least one of the input relations. Note: The two input relations have to have the exact same attributes.

Syntax: $(R \cup S)$.

SQL: `(SELECT * FROM R) UNION ALL (SELECT * FROM S)`

Intersection

Intersection takes in two relations and outputs a relation that contains all tuples that appear in both of the input relations. Note: The two input relations have to have the exact same attributes.

Syntax: $(R \cap S)$.

SQL: `(SELECT * FROM R) INTERSECT (SELECT * FROM S)`

Difference

Difference takes in two relations and outputs a relation that contains all tuples that appear in the first relation but not the second relation. Note: The two input relations have to have the exact same attributes.

Syntax: $(R - S)$.

SQL: `(SELECT * FROM R) EXCEPT (SELECT * FROM S)`

Product

Product takes in two relations and outputs a relation that contains all possible combinations for tuples from the input relations.

Syntax: $(R \times S)$.

SQL: `(SELECT * FROM R) CROSS JOIN (SELECT * FROM S)`, or simply `SELECT * FROM R, S`

Join

Join takes in two relations and outputs a relation that contains all the tuples that are a combination of two tuples where for each attribute that the two relations share, the values for that attribute of both tuples is the same.

Syntax: $(R \bowtie S)$.

SQL: `SELECT * FROM R JOIN S USING (ATTRIBUTE1, ATTRIBUTE2...)`

Observation

Relational algebra is a procedural language because it defines the high level-steps of how to compute a query. For example, $\sigma_{\text{b\_id=102}}(R \bowtie S)$ is saying to first do the join of R and S and then do the select, whereas $(R \bowtie (\sigma_{\text{b\_id=102}}(S)))$ will do the select on S first, and then do the join. These two statements will actually produce the same answer, but if there is only 1 tuple in S with b_id=102 out of a billion tuples, then $(R \bowtie (\sigma_{\text{b\_id=102}}(S)))$ will be significantly faster than $\sigma_{\text{b\_id=102}}(R \bowtie S)$.

A better approach is to say the result you want (retrieve the joined tuples from R and S where b_{id} equals 102), and let the DBMS decide the steps it wants to take to compute the query. SQL will do exactly this, and it is the de facto standard for writing queries on relational model databases.

7 Other Data Models

Document Data Model

A collection of record documents containing a hierarchy of named field/value pairs.

A field's value can be either a scalar type, an array of values, or another document.

Modern implementations use JSON. Older systems use XML or custom object representations.

Vector Data Model

One-dimensional arrays used for nearest-neighbor search (exact or approximate).

Used for semantic search on embeddings generated by ML-trained transformer models (think ChatGPT).

Native integration with modern ML tools and APIs (e.g., LangChain, OpenAI).

At their core, these systems use specialized indexes to perform NN searches quickly.

8 P0 Intro: CRDT

Conflict-free Replicated Data Type(CRDT) is a type of data structure that enables concurrent updates across multiple replicas without the need for coordination between them.

This is useful in scenarios we want a distributed data structure (many copies) that allows local updates to be made independently and the states eventually converge.

Take, for example, a "global" counter that can be incremented independently by each node. These nodes can communicate or "gossip" at any time, sharing their state. The objective is for every node to ultimately reflect the same accurate global value. The key to achieving this with a CRDT lies in its merge function. This method is crafted to be commutative, associative, and idempotent, ensuring reliable data convergence.

Lecture #02: Modern SQL

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 SQL History

Declarative query language for relational databases. It was originally developed in the 1970s as part of the IBM **System R** project. IBM originally called it "SEQUEL" (Structured English Query Language). The name changed in the 1980s to just "SQL" (Structured Query Language).

SQL is not a dead language. It is being updated with new features every couple of years. SQL-92 is the minimum that a DBMS has to support to claim they support SQL. Each vendor follows the standard to a certain degree but there are many proprietary extensions.

Some of the major updates released with each new edition of the SQL standard are shown below.

- **SQL:1999** Regular expressions, Triggers
- **SQL:2003** XML, Windows, Sequences
- **SQL:2008** Truncation, Fancy sorting
- **SQL:2011** Temporal DBs, Pipelined DML
- **SQL:2016** JSON, Polymorphic tables

2 Relational Languages

The language is comprised of different classes of commands:

1. **Data Manipulation Language (DML):** `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.
2. **Data Definition Language (DDL):** Schema definitions for tables, indexes, views, and other objects.
3. **Data Control Language (DCL):** Security, access controls.
4. It also includes view definition, integrity and referential constraints and transactions.

Relational algebra is based on **sets** (unordered, no duplicates). SQL is based on **bags** (unordered, allows duplicates).

important : SQL allows duplicates

3 Aggregates

An aggregation function takes in a bag of tuples as its input and then produces a single scalar value as its output. Aggregate functions can (almost) only be used in a `SELECT` output list.

- `AVG(COL)`: The average of the values in `COL`
- `MIN(COL)`: The minimum value in `COL`
- `MAX(COL)`: The maximum value in `COL`
- `COUNT(COL)`: The number of tuples in the relation

} some sort of aggregation of the selected tuples

Example: *Get # of students with a '@cs' login.*

The following three queries are equivalent:

```sql
CREATE TABLE student (
   sid   INT PRIMARY KEY,
   name  VARCHAR(16),
   login VARCHAR(32) UNIQUE,
   age   SMALLINT,
   gpa   FLOAT
);

CREATE TABLE course (
   cid  VARCHAR(32) PRIMARY KEY,
   name VARCHAR(32) NOT NULL
);

CREATE TABLE enrolled (
   sid   INT REFERENCES student (sid),
   cid   VARCHAR(32) REFERENCES course (cid),
   grade CHAR(1)
);
```

Figure 1: Example database used for lecture

```sql
SELECT COUNT(*) FROM student WHERE login LIKE '%@cs';
```

```sql
SELECT COUNT(login) FROM student WHERE login LIKE '%@cs';
```

```sql
SELECT COUNT(1) FROM student WHERE login LIKE '%@cs';
```

A single SELECT statement can contain multiple aggregates:

Example: *Get # of students and their average GPA with a '@cs' login.*

```sql
SELECT AVG(gpa), COUNT(sid)
  FROM student WHERE login LIKE '%@cs';
```

Some aggregate functions (e.g. COUNT, SUM, AVG) support the DISTINCT keyword:

Example: *Get # of unique students and their average GPA with a '@cs' login.*

```sql
SELECT COUNT(DISTINCT login)
  FROM student WHERE login LIKE '%@cs';
```

Output of other columns outside of an aggregate is undefined (e.cid is undefined below).

Example: *Get the average GPA of students in each course.*

```sql
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid;
```

Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```sql
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
```

```
 WHERE e.sid = s.sid
 GROUP BY e.cid;
```

The HAVING clause filters output results based on aggregation computation. This make HAVING behave like a WHERE clause for a GROUP BY.

Example: *Get the set of courses in which the average student GPA is greater than 3.9.*

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
HAVING avg_gpa > 3.9;
```

The above query syntax is supported by many major database systems, but is not compliant with the SQL standard. To make the query standard compliant, we must repeat use of AVG(S.GPA) in the body of the HAVING clause.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
HAVING AVG(s.gpa) > 3.9;
```

4 String Operations

The SQL standard says that strings are **case sensitive** and **single-quotes only**. There are functions to manipulate strings that can be used in any part of a query.

Pattern Matching: The LIKE keyword is used for string matching in predicates.

- "%" matches any substrings (including empty).
- "_" matches any one character.

String Functions SQL-92 defines string functions. Many database systems implement other functions in addition to those in the standard. Examples of standard string functions include SUBSTRING(S, B, E) and UPPER(S).

Concatenation: Two vertical bars ("||") will concatenate two or more strings together into a single string.

5 Date and Time

Operations to manipulate DATE and TIME attributes. Can be used in either output or predicates. The specific syntax for date and time operations varies wildly across systems.

6 Output Redirection

Instead of having the result a query returned to the client (e.g., terminal), you can tell the DBMS to store the results into another table. You can then access this data in subsequent queries.

- **New Table:** Store the output of the query into a new (permanent) table.

```
SELECT DISTINCT cid INTO CourseIds FROM enrolled;
```

- **Existing Table:** Store the output of the query into a table that already exists in the database. The target table must have the same number of columns with the same types as the target table, but the names of the columns in the output query do not have to match.

```
INSERT INTO CourseIds (SELECT DISTINCT cid FROM enrolled);
```

7 Output Control

Since results SQL are unordered, we must use the `ORDER BY` clause to impose a sort on tuples:

```
SELECT sid, grade FROM enrolled WHERE cid = '15-721'
 ORDER BY grade;
```

The default sort order is ascending (`ASC`). We can manually specify `DESC` to reverse the order:

```
SELECT sid, grade FROM enrolled WHERE cid = '15-721'
 ORDER BY grade DESC;
```

We can use multiple `ORDER BY` clauses to break ties or do more complex sorting:

```
SELECT sid, grade FROM enrolled WHERE cid = '15-721'
 ORDER BY grade DESC, sid ASC;
```

We can also use any arbitrary expression in the `ORDER BY` clause:

```
SELECT sid FROM enrolled WHERE cid = '15-721'
 ORDER BY UPPER(grade) DESC, sid + 1 ASC;
```

By default, the DBMS will return all of the tuples produced by the query. We can use the `LIMIT` clause to restrict the number of result tuples:

```
SELECT sid, name FROM student WHERE login LIKE '%@cs'
 LIMIT 10;
```

We can also provide an offset to return a range in the results:

```
SELECT sid, name FROM student WHERE login LIKE '%@cs'
 LIMIT 20 OFFSET 10;
```

Unless we use an `ORDER BY` clause with a `LIMIT`, the DBMS may produce different tuples in the result on each invocation of the query because the relational model does not impose an ordering.

8 Window Functions

A window function perform "sliding" calculation across a set of tuples that are related. Like an aggregation but tuples are not grouped into a single output tuple.

The conceptual execution for window function is that: 1. the data is partitioned. 2. sort each partition. 3. for each record, it creates a window. 4. it computes the answer for each window.

Functions: The window function can be any of the aggregation functions that we discussed above. There are also also special window functions:

1. ROW_NUMBER: The number of the current row.
2. RANK: The order position of the current row.

RANK() OVER (PARTITION BY cid ORDER BY...)

Grouping: The OVER clause specifies how to group together tuples when computing the window function. Use PARTITION BY to specify group.

```
SELECT cid, sid, ROW_NUMBER() OVER (PARTITION BY cid)
  FROM enrolled ORDER BY cid;
```

We can also put an ORDER BY within OVER to ensure a deterministic ordering of results even if database changes internally.

```
SELECT *, ROW_NUMBER() OVER (ORDER BY cid)
  FROM enrolled ORDER BY cid;
```

IMPORTANT: The DBMS computes RANK after the window function sorting, whereas it computes ROW_NUMBER before the sorting. *** RANK is after the partition sort, ROW_NUMBER is before

Example: *Find the student with the second highest grade for each course.*

```
SELECT * FROM (
  SELECT *, RANK() OVER (PARTITION BY cid
            ORDER BY grade ASC) AS rank
    FROM enrolled) AS ranking
WHERE ranking.rank = 2;
```

9 Nested Queries

Invoke queries inside of other queries to execute more complex logic within a single query. Nested queries are often difficult to optimize.

The scope of outer query is included in an inner query (i.e. the inner query can access attributes from outer query), but not the other way around.

Inner queries can appear in almost any part of a query:

1. SELECT Output Targets:

   ```
   SELECT (SELECT 1) AS one FROM student;
   ```

2. FROM Clause:

   ```
   SELECT name
     FROM student AS s, (SELECT sid FROM enrolled) AS e
     WHERE s.sid = e.sid;
   ```

3. WHERE Clause:

   ```
   SELECT name FROM student
    WHERE sid IN ( SELECT sid FROM enrolled );
   ```

Example: *Get the names of students that are enrolled in '15-445'.*

```
SELECT name FROM student
 WHERE sid IN (
    SELECT sid FROM enrolled
    WHERE cid = '15-445'
 );
```

Note that `sid` has different scope depending on where it appears in the query.

Example: *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT student.sid, name
  FROM student
  JOIN (SELECT MAX(sid) AS sid
          FROM enrolled) AS max_e
    ON student.sid = max_e.sid;
```

Nested Query Results Expressions:

- `ALL`: Must satisfy expression for all rows in sub-query.
- `ANY`: Must satisfy expression for at least one row in sub-query.
- `IN`: Equivalent to `=ANY()`.
- `EXISTS`: At least one row is returned.

Example: *Find all courses that have no students enrolled in it.*

```
SELECT * FROM course
  WHERE NOT EXISTS(
    SELECT * FROM enrolled
      WHERE course.cid = enrolled.cid
);
```

10 Lateral Joins

LATERAL behaves like for loop

The `LATERAL` operator allows a nested query to reference attributes in other nested queries that precede it. (You can think of it like a for loop that allows you to invoke another query for each tuple in a table.)

Example: *Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order..*

```
SELECT * FROM course AS c
  LATERAL (SELECT COUNT(*) AS cnt FROM enrolled
    WHERE enrolled.cid = c.cid) AS t1,
  LATERAL (SELECT AVG(gpa) AS avg FROM student AS s
    JOIN enrolled AS e ON s.sid = e.sid
    WHERE e.cid = c.cid) AS t2;
```

The first lateral compute the number of enrolled students for each course, the second lateral computethe average gpa of enrolled students for each course in the above example.

11 Common Table Expressions

Common Table Expressions (CTEs) are an alternative to windows or nested queries when writing more complex queries. They provide a way to write auxiliary statements for user in a larger query. CTEs can be thought of as a temporary table that is scoped to a single query.

The `WITH` clause binds the output of the inner query to a temporary result with that name.

Example: *Generate a CTE called* `cteName` *that contains a single tuple with a single attribute set to "1". Select all attributes from this CTE.* `cteName`.

```
WITH cteName AS (
    SELECT 1
)
SELECT * FROM cteName;
```

We can bind output columns to names before the `AS`:

```
WITH cteName (col1, col2) AS (
    SELECT 1, 2
)
SELECT col1 + col2 FROM cteName;
```

A single query may contain multiple CTE declarations:

```
WITH cte1 (col1) AS (SELECT 1), cte2 (col2) AS (SELECT 2)
SELECT * FROM cte1, cte2;
```

Adding the `RECURSIVE` keyword after `WITH` allows a CTE to reference itself. This enables the implementation of recursion in SQL queries. With recursive CTEs, SQL is provably Turing-complete, implying that it is as computationally expressive as more general purpose programming languages (if a bit more cumbersome).

Example: *Print the sequence of numbers from 1 to 10.*

```
WITH RECURSIVE cteSource (counter) AS (
    ( SELECT 1 )
    UNION
    ( SELECT counter + 1 FROM cteSource
      WHERE counter < 10 )
)
SELECT * FROM cteSource;
```

Lecture #03: Database Storage (Part I)

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Storage

We will focus on a "disk-oriented" DBMS architecture that assumes that the primary storage location of the database is on non-volatile disk(s).

At the top of the storage hierarchy, you have the devices that are closest to the CPU. This is the fastest storage, but it is also the smallest and most expensive. The further you get away from the CPU, the larger but slower the storage devices get. These devices also get cheaper per GB.

Volatile Devices

- Volatile means that if you pull the power from the machine, then the data is lost.
- Volatile storage supports fast random access with byte-addressable locations. This means that the program can jump to any byte address and get the data that is there.
- For our purposes, we will always refer to this storage class as "memory."

Non-Volatile Devices

- Non-volatile means that the storage device does not require continuous power in order for the device to retain the bits that it is storing.
- It is also block/page addressable. This means that in order to read a value at a particular offset, the program first has to load the 4 KB page into memory that holds the value the program wants to read.
- Non-volatile storage is traditionally better at sequential access (reading multiple contiguous chunks of data at the same time).
- We will refer to this as "disk." We will not make a (major) distinction between solid-state storage (SSD) and spinning hard drives (HDD).

There is also a relatively new class of storage devices that are becoming more popular called *persistent memory*. These devices are designed to be the best of both worlds: almost as fast as DRAM with the persistence of disk. We will not cover these devices in this course, and they are currently not in widespread production use.Probably the most famous example is Optane; unfortunately Intel is winding down its production as of summer 2022. Note that you may see older references to persistent memory as "non-volatile memory".

You may see references to NVMe SSDs, where NVMe stands for non-volatile memory express. These NVMe SSDs are not the same hardware as persistent memory modules. Rather, they are typical NAND flash drives that connect over an improved hardware interface. This improved hardware interface allows for much faster transfers, which leverages improvements in NAND flash perfomance.

Since our DBMS architecture assumes that the database is stored on disk, the components of the DBMS are responsible for figuring out how to move data between non-volatile disk and volatile memory since the system cannot operate on the data directly on disk.

DBMS want to maximize sequential access on non-volatile storage

DBMS must figure out how to move data from disk to memory since it can operate

During this semester, we will refer to DRAM storage as "memory" and anything below that as "disk". We will also not worry about read/writing data to CPU caches or individual CPU registers in this class. We will focus on hiding the latency of the disk rather than optimizations with registers and caches since getting data from disk is so slow. If reading data from the L1 cache reference took one second, reading from an SSD would take 4.4 hours, and reading from an HDD would take 3.3 weeks.

2 Disk-Oriented DBMS Overview

The database is all on disk, and the data in database files is organized into pages, with the first page being the directory page. To operate on the data, the DBMS needs to bring the data into memory. It does this by having a *buffer pool* that manages the data movement back and forth between disk and memory. The DBMS also has an execution engine that will execute queries. The execution engine will ask the buffer pool for a specific page, and the buffer pool will take care of bringing that page into memory and giving the execution engine a pointer to that page in memory. The buffer pool manager will ensure that the page is there while the execution engine operates on that part of memory.

3 DBMS vs. OS

A high-level design goal of the DBMS is to support databases that exceed the amount of memory available. Since reading/writing to disk is expensive, disk use must be carefully managed. We do not want large stalls from fetching something from disk to slow down everything else. We want the DBMS to be able to process other queries while it is waiting to get the data from disk.

This high-level design goal is like virtual memory, where there is a large address space and a place for the OS to bring in pages from disk.

One way to achieve this virtual memory is by using `mmap` to map the contents of a file in a process' address space, which makes the OS responsible for moving pages back and forth between disk and memory. Unfortunately, this means that if `mmap` hits a page fault, the process will be blocked.

- You never want to use `mmap` in your DBMS if you need to write.
- The DBMS (almost) always wants to control things itself and can do a better job at it since it knows more about the data being accessed and the queries being processed.
- The operating system is not your friend.

It is possible to use the OS by using:

- `madvise`: Tells the OS know when you are planning on reading certain pages.
- `mlock`: Tells the OS to not swap memory ranges out to disk.
- `msync`: Tells the OS to flush memory ranges out to disk.

We do not advise using `mmap` in a DBMS for correctness and performance reasons.

Even though the system will have functionalities that seem like something the OS can provide, having the DBMS implement these procedures itself gives it better control and performance.

4 File Storage

In its most basic form, a DBMS stores a database as files on disk. Some may use a file hierarchy, others may use a single file (e.g., SQLite).

The OS does not know anything about the contents of these files. Only the DBMS knows how to decipher their contents, since it is encoded in a way specific to the DBMS.

The DBMS's *storage manager* is responsible for managing a database's files. It represents the files as a collection of pages. It also keeps track of what data has been read and written to pages as well how much free space there is in these pages.

5 Database Pages

The DBMS organizes the database across one or more files in fixed-size blocks of data called *pages*. Pages can contain different kinds of data (tuples, indexes, etc). Most systems will not mix these types within pages. Some systems will require that pages are *self-contained*, meaning that all the information needed to read each page is on the page itself.

Each page is given a unique identifier. If the database is a single file, then the page id can just be the file offset. Most DBMSs have an indirection layer that maps a page id to a file path and offset. The upper levels of the system will ask for a specific page number. Then, the storage manager will have to turn that page number into a file and an offset to find the page.

Most DBMSs uses fixed-size pages to avoid the engineering overhead needed to support variable-sized pages. For example, with variable-size pages, deleting a page could create a hole in files that the DBMS cannot easily fill with new pages.

There are three concepts of pages in DBMS:

1. Hardware page (usually 4 KB).
2. OS page (4 KB).
3. Database page (1-16 KB).

The storage device guarantees an atomic write of the size of the hardware page. If the hardware page is 4 KB and the system tries to write 4 KB to the disk, either all 4 KB will be written, or none of it will. This means that if our database page is larger than our hardware page, the DBMS will have to take extra measures to ensure that the data gets written out safely since the program can get partway through writing a database page to disk when the system crashes.

6 Database Heap

There are a couple of ways to find the location of the page a DBMS wants on the disk, and heap file organization is one of those ways. A *heap file* is an unordered collection of pages where tuples are stored in random order.

The DBMS can locate a page on disk given a `page_id` by using a linked list of pages or a page directory.

1. **Linked List:** Header page holds pointers to a list of free pages and a list of data pages. However, if the DBMS is looking for a specific page, it has to do a sequential scan on the data page list until it finds the page it is looking for.
2. **Page Directory:** DBMS maintains special pages that track locations of data pages along with the amount of free space on each page.

7 Page Layout

Every page includes a header that records meta-data about the page's contents:

- Page size.
- Checksum.
- DBMS version.

- Transaction visibility.
- Self-containment. (Some systems like Oracle require this.)

A strawman approach to laying out data is to keep track of how many tuples the DBMS has stored in a page and then append to the end every time a new tuple is added. However, problems arise when tuples are deleted or when tuples have variable-length attributes.

There are two main approaches to laying out data in pages: (1) slotted-pages and (2) log-structured.

Slotted Pages: Page maps slots to offsets.

- Most common approach used in DBMSs today.
- Header keeps track of the number of used slots, the offset of the starting location of the last used slot, and a slot array, which keeps track of the location of the start of each tuple.
- To add a tuple, the slot array will grow from the beginning to the end, and the data of the tuples will grow from end to the beginning. The page is considered full when the slot array and the tuple data meet.

Log-Structured: Covered in the next lecture.

8 Tuple Layout

A tuple is essentially a sequence of bytes. It is the DBMS's job to interpret those bytes into attribute types and values.

Tuple Header: Contains meta-data about the tuple.

- Visibility information for the DBMS's concurrency control protocol (i.e., information about which transaction created/modified that tuple).
- Bit Map for `NULL` values.
- Note that the DBMS does not need to store meta-data about the schema of the database here.

Tuple Data: Actual data for attributes.

- Attributes are typically stored in the order that you specify them when you create the table.
- Most DBMSs do not allow a tuple to exceed the size of a page.

Unique Identifier:

- Each tuple in the database is assigned a unique identifier.
- Most common: `page_id` + (`offset` or `slot`).
- An application **cannot** rely on these ids to mean anything.

Denormalized Tuple Data: If two tables are related, the DBMS can "pre-join" them, so the tables end up on the same page. This makes reads faster since the DBMS only has to load in one page rather than two separate pages. However, it makes updates more expensive since the DBMS needs more space for each tuple.

Lecture #04: Database Storage (Part II)

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Log-Structured Storage

Some problems associated with the Slotted-Page Design are:

- **Fragmentation**: Deletion of tuples can leave gaps in the pages, making them not fully utilized.
- **Useless Disk I/O**: Due to the block-oriented nature of non-volatile storage, the whole block needs to be fetched to update a tuple.
- **Random Disk I/O**: The disk reader could have to jump to 20 different places to update 20 different tuples, which can be very slow.

What if we were working on a system which only allows creation of new pages and no overwrites? The log-structured storage model works with this assumption and addresses some of the problems listed above.

Log-Structured Storage Overview

Log-structured Storage is based on the Log-Structured File System (LSFS) by Rosenblum and Ousterhout'92 and Log-structured Merge Trees (LSM Tree) by O'Neil, Cheng, and Gawlick'96. Instead of storing tuples, the DBMS only stores the log records of changes to the tuples. The DBMS appends new log entries to an in-memory buffer without checking previous records and then writes out the changes sequentially to disk. Records contain the tuple's unique identifier, the type of operation (PUT/DELETE), and, for put, the contents of the tuple. Effectively, you only keep track of the latest values for each key (most recent PUT/DELETE). Disk writes are sequential and existing pages are immutable which leads to reduced random disk I/O. This is good for append-only storage. To read a record, the DBMS scans the log file backwards from newest to oldest to find the most recent contents of the tuple. To avoid long reads, the DBMS can have indexes (for bookkeeping) to allow it to jump to specific locations in the log.

Compaction

Eventually, the log will get quite big. The DBMS can periodically compact the log by taking only the most recent change for each tuple across several pages. After compaction, the ordering is no longer needed since there's only one of each tuple, so the DBMS can sort by id for faster lookup. These are called Sorted String Tables (SSTables). In **Universal Compaction**, any log files can be compacted together. In **Level Compaction**, the smallest files are level 0. Level 0 files can be compacted to create a bigger level 1 file, level 1 files can be compacted to a level 2 file, etc. **Tiering** is another log compaction method that will not be covered in this course.

Tradeoffs

The tradeoffs of using Log-Structured Storage can be summarized below:

- Fast sequential writes, good for append only storage
- Reads may be slow

- Compaction is expensive
- Subject to write amplification (for each logical write, there could be multiple physical writes).

2 Index-Organized Storage

Observe that both page-oriented storage and log-structured storage rely on additional index to find individual tuples because the tables are inherently unsorted. In the **index-organized storage** scheme, the DBMS directly stores a table's tuples as the value of an index data structure. The DBMS would use a page layout that looks like a slotted page, and tuples are typically sorted in page based on key.

3 Data Representation

The data in a tuple is essentially just byte arrays and doesn't keep track of what kinds of values the attributes are. It is up to the DBMS to know how to keep track of that and interpret those bytes. A *data representation* scheme is how a DBMS stores the bytes for a value.

DBMSs want to make sure the tuples are word-aligned so that the CPU to access it without any unexpected behavior or additional work. Two approaches are usually taken:

- **Padding:** Add empty bits after attributes to ensure that tuple is word aligned.
- **Reordering:** Switch the order of attributes in the physical layout to make sure they are aligned.

There are five high level datatypes that can be stored in tuples: integers, variable-precision numbers, fixed-point precision numbers, variable length values, and dates/times.

Integers

Most DBMSs store integers using their "native" C/C++ types as specified by the IEEE-754 standard. These values are fixed length.

Examples: `INTEGER`, `BIGINT`, `SMALLINT`, `TINYINT`.

Variable Precision Numbers

These are inexact, variable-precision numeric types that use the "native" C/C++ types specified by IEEE-754 standard. These values are also fixed length.

Operations on variable-precision numbers are faster to compute than arbitrary precision numbers because the CPU can execute instructions on them directly. However, there may be rounding errors when performing computations due to the fact that some numbers cannot be represented precisely.

Examples: `FLOAT`, `REAL`.

Fixed-Point Precision Numbers

These are numeric data types with arbitrary precision and scale. They are typically stored in exact, variable-length binary representation (almost like a string) with additional meta-data that will tell the system things like the length of the data and where the decimal should be.

These data types are used when rounding errors are unacceptable, but the DBMS pays a performance penalty to get this accuracy.

Examples: `NUMERIC`, `DECIMAL`.

Variable-Length Data

These represent data types of arbitrary length. They are typically stored with a header that keeps track of the length of the string to make it easy to jump to the next value. It may also contain a checksum for the data.

Most DBMSs do not allow a tuple to exceed the size of a single page. The ones that do store the data on a special "overflow" page and have the tuple contain a reference to that page. These overflow pages can contain pointers to additional overflow pages until all the data can be stored.

Some systems will let you store these large values in an external file, and then the tuple will contain a pointer to that file. For example, if the database is storing photo information, the DBMS can store the photos in the external files rather than having them take up large amounts of space in the DBMS. One downside of this is that the DBMS cannot manipulate the contents of this file. Thus, there are no durability or transaction protections.

Examples: `VARCHAR`, `VARBINARY`, `TEXT`, `BLOB`.

Dates and Times

Representations for date/time vary for different systems. Typically, these are represented as some unit time (micro/milli)seconds since the unix epoch.

Examples: `TIME`, `DATE`, `TIMESTAMP`.

Null Data Types

There are three common approaches to represent nulls in a DBMS.

- **Null Column Bitmap Header:** Store a bitmap in a centralized header that specifies what attributes are null. This is the most common approach.
- **Special Values:** Designate a value to represent `NULL` for a data type (e.g., `INT32_MIN`).
- **Per Attribute Null Flag:** Store a flag that marks that a value is null. This apporach is not recommended because it is not memory-efficient. For each value, the DBMS has to use more than just a single bit to avoid messing up with word alignment.

4 System Catalogs

In order for the DBMS to be able to decipher the contents of tuples, it maintains an internal catalog to tell it meta-data about the databases.

Metadata Contents:

- The tables and columns the database has as well as any indexes on those tables.
- Users of the database and what permissions they have.
- Statistics about the table and what contents are contained within them (i.e., max value of an attribute).

Most DBMSs store their catalog inside of themselves in the format that they use for their tables. They use special code to "bootstrap" these catalog tables.

Lecture #5: Storage Models & Compression

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Database Workloads

OLTP: Online Transaction Processing

An OLTP workload is characterized by fast, short running operations, repetitive operations and simple queries that operate on single entity at a time. OLTP workloads typically handle more writes than reads, and only read/update a small amount of data each time.

An example of an OLTP workload is the Amazon storefront. Users can add things to their cart and make purchases, but the actions only affect their account.

OLAP: Online Analytical Processing

An OLAP workload is characterized by long running, complex queries and reads on large portions of the database. In OLAP workloads, the database system is often analyzing and deriving new data from existing data collected on the OLTP side.

An example of an OLAP workload that analyzes data collected on the OLTP side is personalized Amazon shopping ads. The website analyzes all of data collected from users' carts and purchases, and then selects different ads for different users.

HTAP: Hybrid Transaction + Analytical Processing

A new type of workload (which has become popular recently) is HTAP, where OLTP and OLAP workloads are present together on the same database.

2 Storage Models

There are different ways to store tuples in pages. We have assumed the n-ary storage model so far.

N-Ary Storage Model (NSM)

In the n-ary storage model, the DBMS stores all of the attributes for a single tuple (row) contiguously in a single page. Thus, it is also known as a "row store." This approach is ideal for OLTP workloads where requests are insert-heavy and transactions tend to operate only an individual entity. It is ideal because it takes only one fetch to be able to get all of the attributes for a single tuple.

Advantages:

- Fast inserts, updates, and deletes.
- Good for queries that need the entire tuple.
- Can use index-oriented physical storage for clustering.

Disadvantages:

- Inefficient for scanning large portions of the table and/or a subset of the attributes.
- Poor memory locality for OLAP access patterns.
- Difficult to apply compression because of multiple value domains within a single page.

Decomposition Storage Model (DSM)

DSM aka colstore

In the decomposition storage model, the DBMS stores a single attribute (column) for all tuples contiguously in a block of data. Thus, it is also known as a "column store." This model is ideal for OLAP workloads with many read-only queries that perform large scans over a subset of the table's attributes.

Advantages:

- Reduces the amount of I/O wasted per query because the DBMS only reads the data that it needs for that query.
- Better query processing because of increased locality and cached data reuse.
- Better data compression.

how to put tuples back together in DSM?

Disadvantages:

- Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.

To put the tuples back together when using a column store, there are two common approaches:

- The most commonly used approach is **fixed-length offsets**. Here, the value in a given column will belong to the same tuple as the value in another column at the same offset. Therefore, every single value within the column will have to be the same length.
- A less common approach is to use **embedded tuple ids**. Here, for every attribute in the columns, the DBMS stores a tuple id (ex: a primary key) with it. Then, the system would also store a mapping to tell it how to jump to every attribute that has that id. Note that this method has a large storage overhead because it needs to store a tuple id for every attribute entry.

Partition Attributes Across (PAX)

In the hybrid Partition Attributes Across storage model, the DBMS vertically partitions attributes within a database page. The goal of doing so is to get the benefit of faster processing on columnar storage while retaining the spatial locality benefits of row storage.

In PAX, the rows are horizontally partitioned into groups of rows. Within each row group, the attributes are vertically partitioned into columns. Each row group is similar to a column store for its subset of the rows.

A PAX file has a global header containing a directory with offsets to the file's row groups, and each row group maintains its own header with meta-data about its contents.

PAX is hybrid between row and colstore

3 Database Compression

Compression is widely used in disk-based DBMSs as disk I/O is (almost) always the main bottleneck. It is especially popular in systems that have read-only analytical workloads. The DBMS can fetch more useful tuples if they have been compressed beforehand at the cost of greater computational overhead for compression and decompression.

In-memory DBMSs are more complicated since they do not have to fetch data from disk to execute a query. Memory is much faster than disks, but compressing the database reduces DRAM requirements

and processing. They have to strike a balance between **speed** vs. **compression ratio**. Compressing the database reduces DRAM requirements. It may decrease CPU costs during query execution.

If data sets were completely random bits, there would be no way to perform compression. However, there are key properties of real-world data sets that are amenable to compression:

- Highly *skewed* distributions for attribute values (e.g., Zipfian distribution of the Brown Corpus).
- High *correlation* between attributes of the same tuple (e.g., Zip Code to City, Order Date to Ship Date).

compression should produce fixed length values

Given this, we want a database compression scheme to have the following properties:

- Must produce fixed-length values. The only exception is variable length data stored in separate pools. This is because the DBMS should follow word-alignment and be able to access data using offsets.
- Allow the DBMS to postpone decompression as long as possible during query execution (**late materialization**).
- Must be a **lossless** scheme because people do not like losing data. Any kind of **lossy** compression has to be performed at the application level.

postpone decompression as long as possible : late materialization

must be lossless .
don't want to lose data

Compression Granularity

The kind of data we want to compress greatly affects which compression schemes can be used. There are four levels of compression granularity:

compression at diff granularities

- **Block Level:** Compress a block of tuples for the same table.
- **Tuple Level:** Compress the contents of the entire tuple (NSM only).
- **Attribute Level:** Compress a single attribute value within one tuple. Can target multiple attributes for the same tuple.
- **Columnar Level:** Compress multiple values for one or more attributes stored for multiple tuples (DSM only). This allows for more complicated compression schemes.

4 Naive Compression

The DBMS compresses data using a general purpose algorithm (e.g., gzip, LZO, LZ4, Snappy, Brotli, Oracle OZIP, Zstd). Although there are several compression algorithms that the DBMS could use, engineers often choose ones that often provides lower compression ratio in exchange for faster compression/decompression.

An example of using naive compression is in **MySQL InnoDB**. The DBMS compresses disk pages, pads them to a power of 2KBs and stores them into the buffer pool. However, every time the DBMS tries to read/modify data, the compressed data in the buffer pool must first be decompressed.

Since accessing data requires decompression of compressed data, this limits the scope of the compression scheme. If the goal is to compress the entire table into one giant block, using naive compression schemes would be impossible since the whole table needs to be compressed/decompressed for every access. Therefore, MySQL breaks the table into smaller chunks since the compression scope is limited.

Another problem is that these naive schemes also do not consider the high-level meaning or semantics of the data. The algorithm is oblivious to both the structure of the data, and how the query is planning to access the data. Thus, this eliminates the opportunity to utilize late materialization, since the DBMS cannot tell when it can delay the decompression of data.

naive compression :
must decompress before using

ideally we want the DBMS to operate on compressed data without decompressing first

5 Columnar Compression

comp. at the col level

Run-Length Encoding (RLE)

RLE compresses runs (consecutive instances) of the same value in a single column into triplets:

- The value of the attribute
- The start position in the column segment (offset)
- The number of elements in the run (length)

RLE triplet
(value, offset, length)

The DBMS should sort the columns intelligently beforehand to maximize compression opportunities. This clusters duplicate attributes, thereby increasing the compression ratio. Note that the effectiveness of RLE greatly depends on the underlying data characteristics (e.g. number and frequency of attributes in each data).

Bit-Packing Encoding

When all values for an attribute are less than the value's declared largest size, store them with fewer bits.

Mostly Encoding

Bit-packing variant that uses a special marker to indicate when a value exceeds the largest size and then maintains a look-up table to store them. Use when values are "mostly" less than the largest size.

memory vals that cannot be bit packed are left in raw form

Bitmap Encoding

The DBMS stores a separate bitmap for each unique value of a particular attribute where an offset in the vector corresponds to a tuple. The i^{th} position in the bitmap corresponds to the i^{th} tuple in the table and indicates whether that value is present in the attribute of that tuple. The bitmap is typically segmented into chunks to avoid allocating large blocks of contiguous memory.

This approach is only practical if the value cardinality is low, since the size of the bitmap is linearly proportional to the cardinality of the attribute value. If the cardinality of the value is high, then the bitmap can become larger than the original data set.

only practical if the value cardinality is low

Delta Encoding

Instead of storing exact values, record the difference between values that follow each other in the same column. The base value can be stored in-line or in a separate look-up table. We can also use RLE on the stored deltas to get even better compression ratios.

record the diff between values that follow

Incremental Encoding

This is a type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated. This works best with sorted data.

Dictionary Compression

The most common database compression scheme is dictionary encoding. The DBMS replaces frequent patterns in values with smaller codes. It then stores only these codes and a data structure (i.e. the dictionary) that maps these codes to their original value. A dictionary compression scheme needs to support fast encoding/decoding, as well as range queries.

Encoding and Decoding: The dictionary needs to decide how to **encode** (convert uncompressed value into its compressed form) and **decode** (convert compressed value back into its original form) data. As

such, it is not possible to use hash functions.

The encoded values also need to support sorting in the same order as original values (i.e. **order-preserving encodings**). This ensures that the compressed queries run on compressed data return results that are consistent with uncompressed queries run on the original data. This order-preserving property allows operations to be performed directly on the codes.

Data Structures: The dictionary can be implemented through several data structures:

- **Array:** The dictionary consists of one array of variable length strings and a second array with pointers that maps to string offsets. This is expensive to update so it is only usable in immutable files.
- **Hash Table:** The dictionary as a hash table is fast and compact, but unable to support range and prefix queries.
- **B+Tree:** The dictionary as a B+Tree is slower than a hash table and takes more memory, but it can support range and prefix queries.

Note: Columnar compression schemes work best with read heavy workloads and may need additional support for writes.

Lecture #06: Buffer Pools

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Introduction

The DBMS is responsible for managing its memory and moving data back and forth from the disk. Since, for the most part, data cannot be directly operated on the disk, any database must be able to efficiently move data represented as files from its disk into memory so that it can be used. A diagram of this interaction is shown in Figure 1. Ideally from the execution engine's perspective, it should "appear" as if all the data is in memory. It should not have to worry about how data is fetched into memory or how it is managed. It only requires pointers to memory locations to perform its operations.

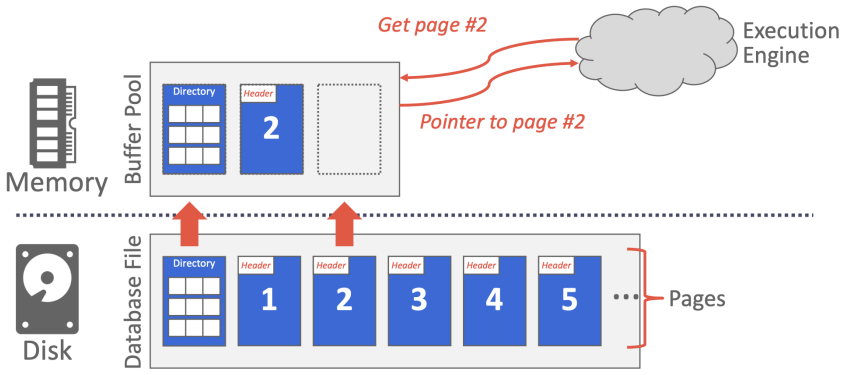


Figure 1: Disk-oriented DBMS.

Another way to think of this problem is in terms of spatial and temporal control.

Spatial Control refers to where pages are physically written on disk. The goal of spatial control is to keep pages that are used together often as physically close together as possible on disk to possibly help with prefetching and other optimizations.

Temporal Control is deciding when to read pages into memory and when to write them to disk. Temporal control aims to minimize the number of stalls from having to read data from disk.

2 Locks vs. Latches

We need to make a distinction between locks and latches when discussing how the DBMS protects its internal elements.

Locks: A lock is a higher-level, logical primitive that protects the contents of a database (e.g., tuples, tables, databases) from other transactions. Database systems can expose to the user which locks are being held as queries are run. Locks need to be able to roll back changes.

Latches: A latch is a low-level protection primitive that the DBMS uses for the critical sections in its internal data structures (e.g., hash tables, regions of memory). Latches are held for only the duration of the operation being made. Latches do not need to be able to roll back changes. This is often implemented by simple language primitives like mutexs and/or conditional variables.

3 Buffer Pool

The *buffer pool* is an in-memory cache of pages read from disk. It is essentially a large memory region allocated inside of the database to temporarily store pages. It is organized as an array of fixed-size pages. Each array entry is called a **frame**. When the DBMS requests a page, it is first searched in the buffer pool, if not found then it is copied from disk into one of the frames. Dirty pages are buffered and not written back immediately (will talk more about this below).

A **page directory** is also maintained on disk, which is the mapping from page IDs to page locations in database files. All changes to the page directory must be recorded on disk to allow the DBMS to find on restart. It is often (not always) kept in memory to minimize latency to page accesses since it has to be read before accessing any physical page.

See Figure 2 for a diagram of the buffer pool's memory organization.

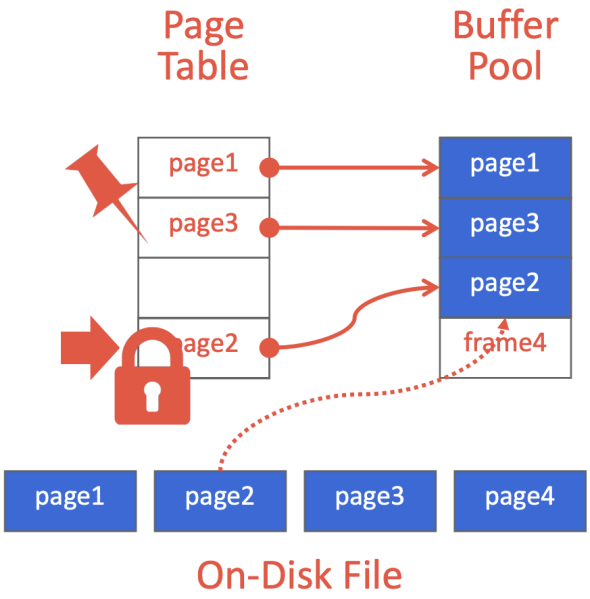


Figure 2: Buffer pool organization and meta-data

Buffer Pool Meta-data

The buffer pool must maintain certain meta-data in order to be used efficiently and correctly.

The **page table** is an in-memory hash table that keeps track of pages that are currently in memory. It maps page IDs to frame locations in the buffer pool. Since the order of pages in the buffer pool does not necessarily reflect the order on the disk, this extra indirection layer allows for the identification of page locations in the pool. The page table also maintains additional meta-data per page, a dirty flag, and a pin/reference counter.

The **dirty-flag** is set by a thread whenever it modifies a page. This indicates to the storage manager that the page must be written back to disk before eviction.

Pin/reference counter tracks the number of threads that are currently accessing that page (either reading or modifying it). A thread has to increment the counter before they access the page. If a page's pin count is greater than zero, then the storage manager is not allowed to evict that page from memory. Pinning does not prevent other transactions from accessing the page concurrently. If the buffer pool runs out of non-pinned pages to evict and the buffer pool is full, an out-of-memory error will be thrown.

Memory Allocation Policies

Memory in the database is allocated for the buffer pool according to two policies.

Global policies deal with decisions that the DBMS should make to benefit the entire workload that is being executed. It considers all active transactions to find an optimal decision for allocating memory.

An alternative is **local policies**, which makes decisions that will make a single query or transaction run faster, even if it isn't good for the entire workload. Local policies allocate frames to a specific transaction without considering the behavior of concurrent transactions. However, it will still support the sharing of frames across transactions.

Most systems use a combination of both global and local policies.

4 Buffer Pool Optimizations

There are several ways to optimize a buffer pool to tailor it to the application's workload.

Multiple Buffer Pools

The DBMS can maintain multiple buffer pools for different purposes (i.e. per-database buffer pool, per-page type buffer pool). Then, each buffer pool can adopt local policies tailored to the data stored inside of it. This method can help reduce latch contention and improve locality.

Object IDs and hashing are two approaches to mapping desired pages to a buffer pool.

Object IDs involve extending the record IDs to have an object identifier. Then through the object identifier, a mapping from objects to specific buffer pools can be maintained. This allows a finer-grained control over buffer pool allocations but has a storage overhead.

Another approach is **hashing** where the DBMS hashes the page ID to select which buffer pool to access. A more generic and uniform approach.

hash to determine which buffer pool

Pre-fetching

The DBMS can also be optimized by pre-fetching pages based on the query plan. Then, while the first set of pages is being processed, the second can be pre-fetched into the buffer pool. This method is commonly used by DBMSs when accessing many pages sequentially like a sequential scan. It is also possible for a buffer pool manager to prefetch leaf pages in a tree index data structure benefiting index scans. **Note:** This need not necessarily be the next physical page on disk but the next logical page in the leaf scan.

next logical page

Scan Sharing (Synchronized Scans)

Query cursors can reuse data retrieved from storage or operator computations. This allows multiple queries to attach to a single cursor that scans a table. If a query starts a scan and there is another active scan, then the DBMS will attach the second query's cursor to the existing cursor. The DBMS keeps track of where the second query joined with the first so that it can finish the scan when it reaches the end of the data structure. This satisfies correctness since the order of scans is not guaranteed by a DBMS and is often useful when a table is frequently scanned.

Note: Continuous scan sharing is an academic prototype that constantly runs a sequential scan for certain tables and queries can hop on and off. However, this is very wasteful and costly when paying per I/O.

Buffer Pool Bypass

The sequential scan operator will not store fetched pages in the buffer pool to avoid overhead. Instead, memory is local to the running query. This works well if an operator needs to read a large sequence of pages that are contiguous on disk and will not be used again. Buffer Pool Bypass can also be used for temporary data (sorting, joins).

memory local to running query

5 Buffer Replacement Policies

When the DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool. To do so it uses a replacement policy to decide which page to evict. The implementation goals of replacement policies are improved correctness, accuracy, speed, and meta-data overhead. **Note:** Remember that a pinned page cannot be evicted.

Least Recently Used (LRU)

The Least Recently Used replacement policy maintains a timestamp of when each page was last accessed. The DBMS picks to evict the page with the oldest timestamp. This timestamp can be stored in a separate data structure, such as a queue, to allow for sorting and improve efficiency by reducing sort time on eviction. However, keeping the data structure sorted and storing a large timestamp has a prohibitive overhead.

CLOCK

The CLOCK policy is an approximation of LRU without needing a separate timestamp per page. In the CLOCK policy, each page is given a reference bit. When a page is accessed, it is set to 1. **Note:** Some implementations may allow an actual ref counter greater than 1.

To visualize this, organize the pages in a circular buffer with a "clock hand". When an eviction is requested, sweep the hand and check if a page's bit is set to 1. If yes, set it to zero, if no, then evict, bring in the new page in its place, and move the hand forward. Additionally, the clock remembers the position between evictions.

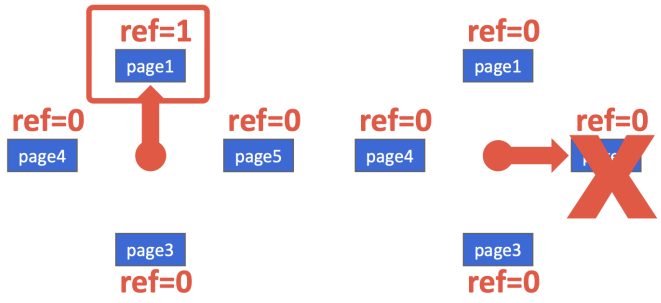


Figure 3: Visualization of CLOCK replacement policy. Page 1 is referenced and set to 1. When the clock hand sweeps, it sets the reference bit for page 1 to 0 and evicts page 5.

LRU & CLOCK both susceptible to sequential flooding

Issues

There are a number of problems with LRU and CLOCK replacement policies.

Firstly, LRU and CLOCK are susceptible to sequential flooding, where the buffer pool's contents are polluted due to a sequential scan. Since sequential scans read many pages quickly, the buffer pool fills up and pages from other queries are evicted as they would have earlier timestamps. In this scenario, the most recent timestamp is not the optimal one to evict.

Secondly, recently used does not account for the frequency of accesses. (E.g. the page directory may be frequently accessed but not recently but it should not be evicted.)

Aternatives

There are three solutions to address the shortcomings of LRU and CLOCK policies.

One solution is **LRU-K** which tracks the history of the last K references as timestamps and computes the interval between subsequent accesses. This history is used to predict the next time a page is going to be accessed. However,

this again has a higher storage overhead. Additionally, need to maintain an in-memory cache of recently evicted pages to prevent thrashing and have some history for recently evicted pages.

An approximation for LRU-2 done by SQL is to have two linked lists and only evict from the old list. Whenever a page is accessed and it is already in the old list put it at the start of the young queue, else put it at the start of the old list.

Another optimization is **localization** per query. The DBMS chooses which pages to evict on a per transaction/query basis. This minimizes the pollution of the buffer pool from each query.

Lastly, **priority hints** allow transactions to tell the buffer pool whether page is important or not based on the context of each page during query execution.

Dirty Pages

There are two methods to handling pages with dirty bits. The fastest option is to drop any page in the buffer pool that is not dirty. A slower method is to write back dirty pages to disk to ensure that its changes are persisted.

These two methods illustrate the trade-off between fast evictions versus dirty writing pages that will not be read again in the future.

One way to avoid the problem of having to write out pages unnecessarily is **background writing**. Through background writing, the DBMS can periodically walk through the page table and write dirty pages to disk. When a dirty page is safely written, the DBMS can either evict the page or just unset the dirty flag.

6 Disk I/O and OS Cache

The DBMS maintains internal queue(s) to track page read/write requests from the entire system. The priority of the tasks are determined based on several factors:

- Sequencial vs. Random I/O
- Critical Path Task vs. Background Task
- Table vs. Index vs. Log vs. Ephemeral Data
- Transaction Information
- User-based SLAs

Most disk operations go through the OS API. Unless explicitly told otherwise, the OS maintains its own filesystem cache.

However, most DBMS use direct I/O (O_DIRECT) to bypass the OS's cache to avoid redundant copies of pages and having to manage different eviction policies. **Postgres** is an example of a database system that uses the OS's Page Cache.

Side Note: *fsync* by default has silent errors and on errors marks the page as clean. OS is not your friend :)

7 Other Memory Pools

The DBMS needs memory for things other than just tuples and indexes. These other memory pools may not always backed by disk depending on implementation.

- Sorting + Join Buffers
- Query Caches
- Maintenance Buffers
- Log Buffers
- Dictionary Caches

Lecture #07: Hash Tables

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Data Structures

A DBMS uses various data structures for many different parts of the system internals. Some examples include:

- **Internal Meta-Data:** This is data that keeps track of information about the database and the system state.
  Ex: Page tables, page directories
- **Core Data Storage:** Data structures are used as the base storage for tuples in the database.
- **Temporary Data Structures:** The DBMS can build ephemeral data structures on the fly while processing a query to speed up execution (e.g., hash tables for joins).
- **Table Indices:** Auxiliary data structures can be used to make it easier to find specific tuples.

There are two main design decisions to consider when implementing data structures for the DBMS:

1. Data organization: We need to figure out how to layout memory and what information to store inside the data structure in order to support efficient access.
2. Concurrency: We also need to think about how to enable multiple threads to access the data structure without causing problems, ensuring that the data remains correct and sound.

2 Hash Table

A hash table implements an associative array abstract data type that maps keys to values. It provides on average $O(1)$ operation complexity ($O(n)$ in the worst-case) and $O(n)$ storage complexity. Note that even with $O(1)$ operation complexity on average, there are constant factor optimizations which are important to consider in the real world.

A hash table implementation is comprised of two parts:

- **Hash Function:** This tells us how to map a large key space into a smaller domain. It is used to compute an index into an array of buckets or slots. We need to consider the trade-off between fast execution and collision rate. On one extreme, we have a hash function that always returns a constant (very fast, but everything is a collision). On the other extreme, we have a "perfect" hashing function where there are no collisions, but would take extremely long to compute. The ideal design is somewhere in the middle.
- **Hashing Scheme:** This tells us how to handle key collisions after hashing. Here, we need to consider the trade-off between allocating a large hash table to reduce collisions and having to execute additional instructions when a collision occurs.

3 Hash Functions

A *hash function* takes in any key as its input. It then returns an integer representation of that key (i.e., the "hash"). The function's output is deterministic (i.e., the same key should always generate the same hash output).

The DBMS need not use a cryptographically secure hash function (e.g., SHA-256) because we do not need to worry about protecting the contents of keys. These hash functions are primarily used internally by the DBMS and thus information is not leaked outside of the system. In general, we only care about the hash function's speed and collision rate.

The current state-of-the-art hash function is Facebook XXHash3.

4 Static Hashing Schemes

A static hashing scheme is one where the size of the hash table is fixed. This means that if the DBMS runs out of storage space in the hash table, then it has to rebuild a larger hash table from scratch, which is very expensive. Typically the new hash table is twice the size of the original hash table.

To reduce the number of wasteful comparisons, it is important to avoid collisions of hashed key. Typically, we use twice the number of slots as the number of expected elements.

The following assumptions usually do not hold in reality:

1. The number of elements is known ahead of time.
2. Keys are unique.
3. There exists a perfect hash function.

Therefore, we need to choose the hash function and hashing schema appropriately.

4.1 Linear Probe Hashing

This is the most basic hashing scheme. It is also typically the fastest. It uses a circular buffer of array slots. The hash function maps keys to slots.

For insertions, when a collision occurs, we linearly search the subsequent slots until an open one is found, looping around from the end to the start of the array if necessary. For lookups, we can check the slot the key hashes to, and search linearly until we find the desired entry. If we reach an empty slot or we iterated over every slot in the hashtable, then the key is not in the table. Note that this means we have to store both key and value in the slot so that we can check if an entry is the desired one.

Deletions are more tricky. We have to be careful about just removing the entry from the slot, as this may prevent future lookups from finding entries that have been put below the now empty slot. There are two solutions to this problem:

- The most common approach is to use "tombstones". Instead of deleting the entry, we replace it with a "tombstone" entry which tells future lookups to keep scanning. Note that insertions are able to insert into tombstone indices.
- The other option is to shift the adjacent data after deleting an entry to fill the now empty slot. However, we must be careful to only move the entries which were originally shifted. This is rarely implemented in practice as it is extremely expensive when we have a large number of keys.

Non-unique Keys: In the case where the same key may be associated with multiple different values or tuples, there are two approaches.

- Separate Linked List: Instead of storing the values with the keys, we store a pointer to a separate storage area that contains a linked list of all the values, which may overflow to multiple pages.
- Redundant Keys: The more common approach is to simply store the same key multiple times in the table. Everything with linear probing still works even if we do this.

Optimizations: There are several ways to further optimize this hashing scheme:

- Specialized hash table implementations based on the data type or size of keys: These could differ in the way they store data, perform splits, etc. For example, if we have string keys, we could store smaller strings in the original hash table and only a pointer or hash for larger strings.
- Storing metadata in a separate array: An example would be to store empty slot/tombstone information in a packed bitmap as a part of the page header or in a separate hash table, which would help us avoid looking up deleted keys.
- Maintaining versions for the hash table and its slots: Since allocating memory for a hash table is expensive, we may want to reuse the same memory repeatedly. To clear out the table and invalidate its entries, we can increment the version counter of the table instead of marking each slot as deleted/empty. A slot can be treated as empty if there is a mismatch between the slot version and table version.

Google's absl::flat_hash_map is a state-of-the-art implementation of Linear Probe Hashing.

4.2 Cuckoo Hashing

Instead of using a single hash table, this approach maintains multiple hashtables with different hash functions. The hash functions are the same algorithm (e.g., XXHash, CityHash); they generate different hashes for the same key by using different seed values.

When we insert, we check every table and choose one that has a free slot (if multiple have one, we can compare things like load factor, or more commonly, just choose a random table). If no table has a free slot, we choose (typically a random one) and evict the old entry. We then rehash the old entry into a different table. In rare cases, we may end up in a cycle. If this happens, we can rebuild all of the hash tables with new hash function seeds (less common) or rebuild the hash tables using larger tables (more common).

Cuckoo hashing guarantees $O(1)$ lookups and deletions, but insertions may be more expensive.

Professor's note: The essence of cuckoo hashing is that multiple hash functions map a key to different slots. In practice, cuckoo hashing is implemented with multiple hash functions that map a key to different slots in a single hash table. Further, as hashing may not always be $O(1)$, cuckoo hashing lookups and deletions may cost more than $O(1)$.

5 Dynamic Hashing Schemes

The static hashing schemes require the DBMS to know the number of elements it wants to store. Otherwise it has to rebuild the table if it needs to grow/shrink in size.

Dynamic hashing schemes are able to resize the hash table on demand without needing to rebuild the entire table. The schemes perform this resizing in different ways that can either maximize reads or writes.

5.1 Chained Hashing

This is the most common dynamic hashing scheme. The DBMS maintains a linked list of buckets for each slot in the hash table. Keys which hash to the same slot are simply inserted into the linked list for that slot.

To look up an element, we hash to its bucket and then scan for it.

Bloom Filters: Lookup can be optimized by additionally storing bloom filters in the bucket pointer list, which would tell us if a key does not exist in the linked list, thus helping us avoid the cost of lookup in such cases. The bloom filter is a probabilistic data structure, a bitmap, that answers set membership queries. It allows false positives but never false negatives.

Assume we use bitmaps of size n and we employ k hash functions $h_1, h_2, ..., h_k$ to implement the bloom filters. Adding bloom filters modifies the hashing implementation in the following ways:

- Insertion: When we insert x into a bucket in our hash table, we set the indices $h_1(x)\%n$, $h_2(x)\%n$, ..., $h_k(x)\%n$ of the bucket's bitmap as 1s.
- Lookup: When we lookup x, which hashes to a certain bucket, we check whether all indices $h_1(x)\%n$, $h_2(x)\%n$, ..., $h_k(x)\%n$ of the bucket's bitmap are 1s. If they are, the bloom filter tells us that x *may* exist in the bucket linked list. If not, the bloom filter tells us that x *definitely* does not exist in the bucket linked list.

5.2 Extendible Hashing

Improved variant of chained hashing that splits buckets instead of letting chains to grow forever. This approach allows multiple slot locations in the hash table to point to the same bucket chain.

The core idea behind re-balancing the hash table is to to move bucket entries on split and increase the number of bits to examine to find entries in the hash table. This means that the DBMS only has to move data within the buckets of the split chain; all other buckets are left untouched.

- The DBMS maintains a global and local depth bit counts. These bit counts determine the number of most significant bits we need to look at to find buckets in the slot array.
- When a bucket is full, the DBMS splits the bucket and re-distributes its elements. If the local depth of the split bucket is less than the global depth, then the new bucket is just added to the existing slot array. Otherwise, the DBMS doubles the size of the slot array to accommodate the new bucket and increments the global depth counter.

5.3 Linear Hashing

Instead of immediately splitting a bucket when it overflows, this scheme maintains a *split pointer* that keeps track of the next bucket to split. No matter whether this pointer is pointing to the bucket that overflowed, the DBMS always splits. The overflow criterion is left up to the implementation.

- When any bucket overflows, split the bucket at the pointer location. Add a new slot entry and a new hash function, and apply this function to rehash the keys in the split bucket.
- If the original hash function maps to a slot that has previously been pointed to by the split pointer, apply the new hash function to determine the actual location of the key.
- When the pointer reaches the very last slot, delete the original hash function and move the pointer back to the beginning.

If the highest bucket below the split pointer is empty, we can also remove the bucket and move the split pointer in the reverse direction, thereby shrinking the size of the hash table.

Lecture #08: Tree Indexes

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Table Indexes

There are a number of different data structures one can use inside of a database system for purposes such as internal meta-data, core data storage, temporary data structures, or table indexes. For table indexes, which may involve queries with range scans, a hash table may not be the best option since it's inherently unordered.

A *table index* is a replica of a subset of a table's columns that is organized and/or sorted for efficient access using a subset of those attributes. So instead of performing a sequential scan, the DBMS can perform a lookup on the table index to find certain tuples more quickly. The DBMS ensures that the contents of the tables and the indexes are always logically in sync.

There exists a trade-off between the number of indexes to create per database. Although more indexes makes looking up queries faster, indexes also use storage and require maintenance. Plus, there are concurrency concerns with respect to keeping them in sync. It is the DBMS's job to figure out the best indexes to use to execute queries.

2 B+Tree

A *B+Tree* is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertion, and deletions in $O(log(n))$. It is optimized for disk-oriented DBMS's that read/write large blocks of data.

Almost every modern DBMS that supports order-preserving indexes uses a B+Tree. There is a specific data structure called a *B-Tree*, but people also use the term to generally refer to a class of data structures. The primary difference between the original *B-Tree* and the B+Tree is that B-Trees stores keys and values in *all nodes*, while B+ trees store values *only in leaf nodes*. Modern B+Tree implementations combine features from other B-Tree variants, such as the sibling pointers used in the B^{link}-Tree.

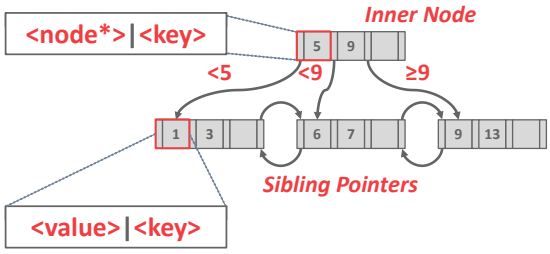


Figure 1: B+ Tree diagram

Formally, a B+Tree is an M-way search tree (where M represents the maximum number of children a node can have) with the following properties:

- It is perfectly balanced (i.e., every leaf node is at the same depth).
- Every inner node other than the root is at least half full ($M/2 - 1 <=$ num of keys $<= M - 1$).
- Every inner node with k keys has k+1 non-null children.

Every node in a B+Tree contains an array of key/value pairs.

For leaf nodes, the keys are derived from the attribute(s) that the index is based on. Though it is not necessary according to the definition of the B+Tree, arrays at every node are almost always sorted by the keys. Two approaches for leaf node values are *record IDs* and *tuple data*. Record IDs refer to a pointer to the location of the tuple, usually the primary key. Leaf nodes that have tuple data store the the actual contents of the tuple in each node.

For inner nodes, the values contain pointers to other nodes, and the keys can be thought of as guide posts. They guide the tree traversal but do not represent the keys (and hence their values) on the leaf nodes. What this means is that you could potentially have a key in an inner node (as a guide post) that is not found on the leaf nodes. Although it must be noted that conventionally inner nodes posses only those keys that are present in the leaf nodes.

Depending on the index type (NULL first, or NULL last), null keys will be clustered in either the first leaf node or the last leaf node.

Insertion

To insert a new entry into a B+Tree, one must traverse down the tree and use the inner nodes to figure out which leaf node to insert the key into.

1. Find correct leaf L.
2. Add new entry into L in sorted order:
   - If L has enough space, the operation is done.
   - Otherwise split L into two nodes L and L_2. Redistribute entries evenly and copy up the middle key. Insert an entry pointing to L_2 into the parent of L.
3. To split an inner node, redistribute entries evenly, but push up the middle key.

Deletion

Whereas in inserts we occasionally had to split leaves when the tree got too full, if a deletion causes a tree to be less than half-full, we must merge in order to re-balance the tree.

1. Find correct leaf L.
2. Remove the entry:
    - If L is at least half full, the operation is done.
    - Otherwise, you can try to redistribute, borrowing from the sibling.
    - If redistribution fails, merge L and the sibling.
3. If a merge occurred, you must delete the entry in the parent pointing to L.

Composite Index

The key is composed of multiple attributes.

We can create composite index like:

```
CREATE INDEX abc_index ON table (a, b, c);
```

Then we can use the index for queries like:

```
SELECT a, b, c FROM table
WHERE a = 1 AND b = 2 AND c = 3;
```

Note that the 'AND' operator is mostly necessary. Conditions with 'OR' are generally not supported.

Selection Conditions

Because B+Trees are in sorted order, look ups have fast traversal and also do not require the entire key. The DBMS can use a B+Tree index if the query provides any of the attributes of the search key. This differs from a hash index, which requires all attributes in the search key.

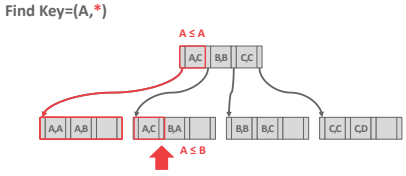


Figure 2: To perform a prefix search on a B+Tree, one looks at the first attribute on the key, follows the path down and performs a sequential scan across the leaves to find all they keys that one wants.

Duplicate Keys

There are two approaches to duplicate keys in a B+Tree.

The first approach is to *append record IDs* as part of the key. Since each tuple's record ID is unique, this will ensure that all the keys are identifiable.

The second approach is to allow leaf nodes to spill into *overflow nodes* that contain the duplicate keys. Although no redundant information is stored, this approach is more complex to maintain and modify.

Clustered Indexes

The table is stored in the sort order specified by the primary key, as either heap- or index-organized storage. Since some DBMSs always use a clustered index, they will automatically make a hidden row id primary key if a table doesn't have an explicit one, but others cannot use them at all.

Heap Clustering

Tuples are sorted in the heap's pages using the order specified by a clustering index. DBMS can jump directly to the pages if clustering index's attributes are used to access tuples.

Index Scan Page Sorting

Since directly retrieving tuples from an unclustered index is inefficient, the DBMS can first figure out all the tuples that it needs and then sort them based on their page id. This way, each page will only need to be fetched exactly once.

3 B+Tree Design Choices

3.1 Node Size

Depending on the storage medium, we may prefer larger or smaller node sizes. For example, nodes stored on hard drives are usually on the order of megabytes in size to reduce the number of seeks needed to find data and amortize the expensive disk read over a large chunk of data, while in-memory databases may use page sizes as small as 512 bytes in order to fit the entire page into the CPU cache as well as to decrease data fragmentation. This choice can also be dependent on the type of workload, as point queries would prefer as small a page as possible to reduce the amount of unnecessary extra info loaded, while a large sequential scan might prefer large pages to reduce the number of fetches it needs to do.

3.2 Merge Threshold

While B+Trees have a rule about merging underflowed nodes after a delete, sometimes it may be beneficial to temporarily violate the rule to reduce the number of deletion operations. For instance, eager merging could lead to thrashing, where a lot of successive delete and insert operations lead to constant splits and merges. It also allows for batched merging where multiple merge operations happen all at once, reducing the amount of time that expensive write latches have to be taken on the tree.

There are merge strategy that keeps small nodes in the tree and rebuilds it later, which made the tree unbalanced (as in Postgres). We will not discuss this in the lecture.

3.3 Variable Length Keys

Currently we have only discussed B+Trees with fixed length keys. However we may also want to support variable length keys, such as the case where a small subset of large keys lead to a lot of wasted space. There are several approaches to this:

1. **Pointers**
   Instead of storing the keys directly, we could just store a pointer to the key. Due to the inefficiency of having to chase a pointer for each key, the only place that uses this method in production is embedded devices, where its tiny registers and cache may benefit from such space savings
2. **Variable Length Nodes**
   We could also still store the keys like normal and allow for variable length nodes. This is generally infeasible and largely not used due to the significant memory management overhead of dealing with variable length nodes.
3. **Padding**
   Instead of varying the key size, we could set each key's size to the size of the maximum key and pad out all the shorter keys. In most cases this is a massive waste of memory, so you don't see this used by anyone either.
4. **Key Map/Indirection**
   The method that nearly everyone uses is replacing the keys with an index to the key-value pair in a separate dictionary. This offers significant space savings and potentially shortcuts point queries (since the key-value pair the index points to is the exact same as the one pointed to by leaf nodes). Due to the small size of the dictionary index value, there is enough space to place a prefix of each key alongside the index, potentially allowing some index searching and leaf scanning to not even have to chase the pointer (if the prefix is at all different from the search key).

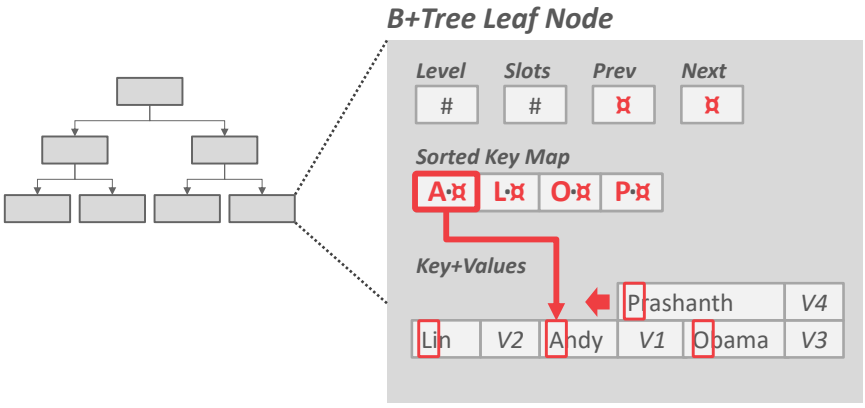


Figure 3: An example of Key Map/Indirection. The map stores a small prefix of the key, as well as a pointer to the key-value pair.

3.4 Intra-Node Search

Once we reach a node, we still need to search within the node (either to find the next node from an inner node, or to find our key value in a leaf node). While this is relatively simple, there are still some tradeoffs to consider:

1. **Linear**
   The simplest solution is to just scan every key in the node until we find our key. On the one hand, we don't have to worry about sorting the keys, making insertions and deletes much quicker. On the other hand, this is relatively inefficient and has a complexity of $\mathbb{O}(n)$ per search. This can be vectorized using SIMD (or equivalent) instructions.
2. **Binary**
   A more efficient solution for searching would be to keep each node sorted and use binary search to find the key. This is as simple as jumping to the middle of a node and pivoting left or right depending on the comparison between the keys. Searches are much more efficient this way, as this method only has the complexity of $\mathbb{O}(\ln(n))$ per search. However, insertions become more expensive as we must maintain the sort of each node.
3. **Interpolation**
   Finally, in some circumstances we may be able to utilize interpolation to find the key. This method takes advantage of any metadata stored about the node (such as max element, min element, average, etc.) and uses it to generate an approximate location of the key. For example, if we are looking for 8 in a node and we know that 10 is the max key and $10 - (n + 1)$ is the smallest key (where n is the number of keys in each node), then we know to start searching 2 slots down from the max key, as the key one slot away from the max key must be 9 in this case. Despite being the fastest method we have given, this method is only seen in academic databases due to its limited applicability to keys with certain properties (like integers) and complexity.

4 Optimizations

4.1 Prefix Compression

Most of the time when we have keys in the same node there will be some partial overlap of some prefix of each key (as similar keys will end up right next to each other in a sorted B+Tree). Instead of storing this prefix as part of each key multiple times, we can simply store the prefix once at the beginning of the node and then only include the unique sections of each key in each slot.

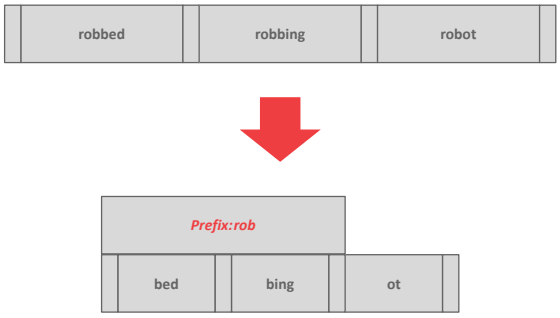


Figure 4: An example of prefix compression. Since the keys are in lexicographic order, they are likely to share some prefix.

4.2 Deduplication

In the case of an index which allows non-unique keys, we may end up with leaf nodes containing the same key over and over with different values attached. One optimization of this could be only writing the key once and then following it with all of its associated values.

4.3 Suffix Truncation

For the most part the key entries in inner nodes are just used as signposts and not for their actual key value (as even if a key exists in the index we still have to search to the bottom to ensure that it hasn't been deleted). We can take advantage of this by only storing the minimum prefix that is needed to correctly route probes into the correct node.

4.4 Pointer Swizzling

Because each node of a B+Tree is stored in a page from the buffer pool, each time we load a new page we need to fetch it from the buffer pool, requiring latching and lookups. To skip this step entirely, we could store the actual raw pointers in place of the page IDs (known as "swizzling"), preventing a buffer pool fetch entirely. Rather than manually fetching the entire tree and placing the pointers manually, we can simply store the resulting pointer from a page lookup when traversing the index normally. Note that we must track which pointers are swizzled and deswizzle them back to page ids when the page they point to is unpinned and victimized.

4.5 Bulk Insert

When a B+Tree is initially built, having to insert each key the usual way would lead to constant split operations. Since we already give leaves sibling pointers, initial insertion of data is much more efficient if we construct a sorted linked list of leaf nodes and then easily build the index from the bottom up using the first key from each leaf node. Note that depending on our context we may wish to pack the leaves

as tightly as possible to save space or leave space in each leaf to allow for more inserts before a split is necessary.

4.6 Write-Optimized B+ Tree

Split / merge node operation are expensive. Therefore, some variants of B-Tree, such as Bϵ-Tree, logs changes in the internal node and lazily propagates the updates down to the leaf node later.

Lecture #09: Index Concurrency Control

15-445/645 Database Systems (Spring 2024)

https://15445.courses.cs.cmu.edu/spring2024/

Carnegie Mellon University
Jignesh Patel

1 Index Concurrency Control

So far, we assumed that the data structures we have discussed are single-threaded. However, most DBMSs needs to allow multiple threads to safely access data structures to take advantage of additional CPU cores and hide disk I/O stalls.

There are systems using single-thread model. Also an easy way to convert a single-thread data structure to a multi-thread one is to use a read-write lock, but it is not efficient.

A *concurrency control* protocol is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.

A protocol's correctness criteria can vary:

- **Logical Correctness:** This means that the thread is able to read values that it should expects to read, e.g. a thread should read back the value it had written previously.
- **Physical Correctness:** This means that the internal representation of the object is sound, e.g. there are not pointers in the data structure that will cause a thread to read invalid memory locations.

For the purposes of this lecture, we only care about enforcing physical correctness. We will revisit logical correctness in later lectures.

2 Locks vs. Latches

There is an important distinction between locks and latches when discussing how the DBMS protects its internal elements.

Locks

A lock is a higher-level, logical primitive that protects the contents of a database (e.g., tuples, tables, databases) from other transactions. Transactions will hold a lock for its entire duration. Database systems can expose to the user the locks that are being held as queries are run. There should be some higher-level mechanism to detect deadlocks and rollback changes.

Latches

Latches are the low-level protection primitives used for critical sections the DBMS's internal data structures (e.g., data structure, regions of memory) from other threads. Latches are held for a short period for a simple operation in a database system (i.e., page latch). There are two modes for latches:

- **`READ`:** Multiple threads are allowed to read the same item at the same time. A thread can acquire the latch in read mode even if another thread has already acquired it in read mode.
- **`WRITE`:** Only one thread is allowed to access the item. A thread cannot acquire a write latch if another thread holds the latch in any mode. A thread holding a write latch also prevents other threads from acquiring a read latch.

3 Latch Implementations

Latch implementation should have a small memory footprint, and may have a fast path to acquire the latch when there is no contention.

The underlying primitive that used to implement a latch is through atomic instructions that modern CPUs provide. With this, a thread can check the contents of a memory location to see whether it has a certain value.

- **Atomic Instruction Example: compare-and-swap (CAS)**
  Atomic instruction that compares contents of a memory location M to a given value V
  - If values are equal, installs new given value V' in M
  - Otherwise, operation fails

There are several approaches to implementing a latch in a DBMS. Each approach has different trade-offs in terms of engineering complexity and runtime performance. These test-and-set steps are performed atomically (i.e., no other thread can update the value in between the test and set steps.

Test-and-Set Spin Latch (TAS)

Spin latches are a more efficient alternative to an OS mutex as it is controlled by the DBMSs. A spin latch is essentially a location in memory that threads try to update (e.g., setting a boolean value to true). A thread performs CAS to attempt to update the memory location. The DBMS can control what happens if it fails to get the latch. It can choose to try again (for example, using a while loop) or allow the OS to deschedule it. Thus, this method gives the DBMS more control than the OS mutex, where failing to acquire a latch gives control to the OS.

- **Example:** `std::atomic<T>`
- **Advantages:** Latch/unlatch operations are efficient (single instruction to lock/unlock on x86).
- **Disadvantages:** Not scalable nor cache-friendly because with multiple threads, the CAS instructions will be executed multiple times in different threads. These wasted instructions will pile up in high contention environments; the threads look busy to the OS even though they are not doing useful work. This leads to cache coherence problems because threads are polling cache lines on other CPUs.

Blocking OS Mutex

One possible implementation of latches is the OS built-in mutex infrastructure. Linux provides the futex (fast user-space mutex), which is comprised of (1) a spin latch in user-space and (2) an OS-level mutex. If the DBMS can acquire the user-space latch, then the latch is set. It appears as a single latch to the DBMS even though it contains two internal latches. If the DBMS fails to acquire the user-space latch, then it goes down into the kernel and tries to acquire a more expensive mutex. If the DBMS fails to acquire this second mutex, then the thread notifies the OS that it is blocked on the mutex and then it is de-scheduled.

OS mutex is generally a bad idea inside of DBMSs as it is managed by OS and has large overhead.

- **Example:** `std::mutex`
- **Advantages:** Simple to use and requires no additional coding in DBMS.
- **Disadvantages:** Expensive and non-scalable (about 25 ns per lock/unlock invocation) because of OS scheduling.

Reader-Writer Latches

Mutexes and Spin Latches do not differentiate between reads/writes (i.e., they do not support different modes). The DBMS needs a way to allow for concurrent reads, so if the application has heavy reads it will have better performance because readers can share resources instead of waiting.

A Reader-Writer Latch allows a latch to be held in either read or write mode. It keeps track of how many threads hold the latch and are waiting to acquire the latch in each mode. Reader-writer latches use one of the previous two latch implementations as primitives and have additional logic to handle reader-writer queues, which are queues requests for the latch in each mode. Different DBMSs can have different policies for how it handles the queues.

One thing to notice is that different reader-writer lock implementations have different waiting policies. There are reader-preferred, writer-preferred, and fair reader-writer locks. The behavior differs in different operating systems and pthread implementations.

- **Example:** `std::shared_mutex`
- **Advantages:** Allows for concurrent readers.
- **Disadvantages:** The DBMS has to manage read/write queues to avoid starvation. Larger storage overhead than spin Latches due to additional meta-data.

4 Hash Table Latching

It is easy to support concurrent access in a static hash table due to the limited ways threads access the data structure. For example, all threads move in the same direction when moving from slot to the next (i.e., top-down). Threads also only access a single page/slot at a time. Thus, deadlocks are not possible in this situation because no two threads could be competing for latches held by the other. When we need to resize the table, we can just take a global latch on the entire table to perform the operation.

Latching in a dynamic hashing scheme (e.g., extendible) is a more complicated scheme because there is more shared state to update, but the general approach is the same.

There are two approaches to support latching in a hash table that differ on the granularity of the latches:

- **Page Latches:** Each page has its own Reader-Writer latch that protects its entire contents. Threads acquire either a read or write latch before they access a page. This decreases parallelism because potentially only one thread can access a page at a time, but accessing multiple slots in a page will be fast for a single thread because it only has to acquire a single latch.
- **Slot Latches:** Each slot has its own latch. This increases parallelism because two threads can access different slots on the same page. But it increases the storage and computational overhead of accessing the table because threads have to acquire a latch for every slot they access, and each slot has to store data for the latches. The DBMS can use a single mode latch (i.e., Spin Latch) to reduce meta-data and computational overhead at the cost of some parallelism.

It is also possible to create a latch-free linear probing hash table directly using compare-and-swap (CAS) instructions. Insertion at a slot can be achieved by attempting to compare-and-swap a special "null" value with the tuple we wish to insert. If this fails, we can probe the next slot, continuing until it succeeds.

5 B+Tree Latching

The challenge of B+Tree latching is preventing the two following problems:

- Threads trying to modify the contents of a node at the same time.

- One thread traversing the tree while another thread splits/merges nodes.

Latch crabbing/coupling is a protocol to allow multiple threads to access/modify B+Tree at the same time. The basic idea is as follows.

1. Get latch for the parent.
2. Get latch for the child.
3. Release latch for the parent if the child is deemed "safe". A "safe" node is one that will not split, merge, or redistribute when updated. In other words, a node is "safe" if
   - **for insertion**: it is not full.
   - **for deletion**: it is more than half full.

Note that read latches do not need to worry about the "safe" condition.

Basic Latch Crabbing Protocol:

- **Search:** Start at the root and go down, repeatedly acquire latch on the child and then unlatch parent.
- **Insert/Delete:** Start at the root and go down, obtaining `X` latches as needed. Once the child is latched, check if it is safe. If the child is safe, release latches on all its ancestors.

The order in which latches are released is not important from a correctness perspective. However, from a performance point of view, it is better to release the latches that are higher up in the tree since they block access to a larger portion of leaf nodes.

Improved Latch Crabbing Protocol: The problem with the basic latch crabbing algorithm is that transactions **always** acquire an exclusive latch on the root for every insert/delete operation. This limits parallelism. Instead, one can assume that having to resize (i.e., split/merge nodes) is rare, and thus transactions can acquire shared latches down to the leaf nodes. Each transaction will assume that the path to the target leaf node is safe, and use `READ` latches and crabbing to reach it and verify. If the leaf node is not safe, then we abort and do the previous algorithm where we acquire `WRITE` latches.

- **Search:** Same algorithm as before.
- **Insert/Delete:** Set `READ` latches as if for search, go to leaf, and set `WRITE` latch on leaf. If the leaf is not safe, release all previous latches, and restart the transaction using previous Insert/Delete protocol.

Leaf Node Scans

The threads in these protocols acquire latches in a "top-down" manner. This means that a thread can only acquire a latch from a node that is below its current node. If the desired latch is unavailable, the thread must wait until it becomes available. Given this, there can never be deadlocks.

However, leaf node scans are susceptible to deadlocks because now we have threads trying to acquire exclusive locks in two different directions at the same time (e.g., thread 1 tries to delete, thread 2 does a leaf node scan). Index latches do not support deadlock detection or avoidance.

Thus, the only way programmers can deal with this problem is through coding discipline. The leaf node sibling latch acquisition protocol must support a "no-wait" mode. That is, the B+tree code must cope with failed latch acquisitions. Since latches are intended to be held (relatively) briefly, if a thread tries to acquire a latch on a leaf node but that latch is unavailable, then it should abort its operation (releasing any latches that it holds) quickly and restart the operation.

Lecture #10: Sorting & Aggregation Algorithms

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Query Plan

Up to this point we have talked about access methods. Now we need to actually execute the queries.

The database system will compile SQLs into query plans. The query plan is a tree of operators. We will cover this later in query execution lectures.

For a disk-oriented database system, we will use the buffer pool to implement algorithms that need to spill to disk. We want to minimize I/O for an algorithm.

2 Sorting

DBMSs need to sort data because tuples in a table have no specific order under the relation model. Sorting is (potentially) used in `ORDER BY`, `GROUP BY`, `JOIN`, and `DISTINCT` operators. If the data that needs to be sorted fits in memory, then the DBMS can use a standard sorting algorithm (e.g., quicksort). If the data does not fit, then the DBMS needs to use external sorting that is able to spill to disk as needed and prefers sequential over random I/O.

Top-N Heap Sort: If a query contains an `ORDER BY` with a `LIMIT`, then to find the top-N elements, the DBMS only needs to scan the data once while maintaining a priority queue of the top-N elements it has seen so far. The ideal scenario for heapsort is when the top-N elements fit in memory, so the DBMS can maintain the entire priority queue in-memory.

External merge sort: This is the standard algorithm for sorting data which is too large to fit in memory. It is a divide-and-conquer sorting algorithm that splits the data set into separate *runs* and then sorts them individually. It can spill runs to disk as needed then read them back in one at a time. The algorithm is comprised of two phases:

- **Phase #1 – Sorting:** First, the algorithm sorts small chunks of data that fit in main memory, and then writes the sorted pages back to disk.
- **Phase #2 – Merge:** Then, the algorithm combines the sorted runs into larger sorted runs.

A sorted run can be early-materialized, which means that the entire tuple is stored in the pages, or can be late-materialized, which means we only store record IDs in the pages and read them later.

Two-way Merge Sort

The most basic version of the external merge sort algorithm is the two-way merge sort. During the sorting phase, the algorithm reads each page, sorts it, and writes the sorted version back to disk. Then, in the merge phase, it uses three buffer pages. It reads two sorted pages in from disk, and merges them together into a third buffer page. Whenever the third page fills up, it is written back to disk and replaced with an empty page. Each set of sorted pages is called a *run*. The algorithm then recursively merges the runs together.

Let N be the total number of data pages. The algorithm makes $1 + \lceil log_2 N \rceil$ total passes through the data (1 for the first sorting step then $\lceil log_2 N \rceil$ for the recursive merging). The total I/O cost is $2N \times$ (# of passes) since each pass performs an I/O read and an I/O write for each page.

General (K-way) Merge Sort

The generalized version of the algorithm allows the DBMS to take advantage of using more than three buffer pages. Let B be the total number of buffer pages available. Then, during the sort phase, the algorithm can read and sort B pages at a time, so the DBMS writes $\lceil \frac{N}{B} \rceil$ sorted runs of length B pages back to disk. The merge phase can combine up to $B - 1$ runs in each pass, using one buffer page per run to read it and again using one buffer page for the combined data, writing back to disk as needed.

In the generalized version, the algorithm performs $1 + \lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil$ passes (one for the sorting phase and $\lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil$ for the merge phase. Then, the total I/O cost is $2N \times$ (# of passes) since it again has to make a read and write for each page in each pass.

Double Buffering Optimization

One optimization for external merge sort is prefetching the next run in the background and storing it in a second buffer while the system is processing the current run. This reduces the wait time for I/O requests at each step by continuously utilizing the disk. This optimization requires the use of multiple threads, since the prefetching should occur while the computation for the current run is happening.

Comparison Optimizations

Code specialization is often used to speedup sorting comparisions. Instead of providing the comparator as a function pointer to the sorting algorithm, the sorting function can be hard-coded to the specific key type. An example of this is template specialization in C++. Another optimization (for string based comparisons) is **suffix truncation**, wherein the binary prefix of long VARCHAR keys can be compared for equality checks with a fallback to the slower string comparison if the prefixes are equal.

Using B+Trees

It is sometimes advantageous for the DBMS to use an existing B+tree index to aid in sorting rather than using the external merge sort algorithm. In particular, if the index is a clustered index, the DBMS can just traverse the B+tree. Since the index is clustered, the data will be stored in the correct order, so the I/O access will be sequential. This means it is always better than external merge sort since no computation is required. On the other hand, if the index is unclustered, traversing the tree is almost always worse, since each record could be stored in any page, so nearly all record accesses will require a disk read.

3 Aggregations

An aggregation operator in a query plan collapses the values of one or more tuples into a single scalar value. There are two approaches for implementing an aggregation: (1) sorting and (2) hashing.

Sorting

The DBMS first sorts the tuples on the `GROUP BY` key(s). It can use either an in-memory sorting algorithm if everything fits in the buffer pool (e.g., quicksort) or the external merge sort algorithm if the size of the data exceeds memory. The DBMS then performs a sequential scan over the sorted data to compute the aggregation. The output of the operator will be sorted on the keys.

When performing sorting aggregations, it is important to order the query operations to maximize efficiency. For example, if the query requires a filter, it is better to perform the filter first and then sort the filtered data to reduce the amount of data that needs to be sorted.

Hashing

Hashing can be computationally cheaper than sorting for computing aggregations, especially when the order of the output does not matter. The DBMS populates an ephemeral hash table as it scans the table. For each record, check whether there is already an entry in the hash table and perform the appropriate modification. If the size of the hash table is too large to fit in memory, then the DBMS has to spill it to disk. There are two phases to accomplish this:

- **Phase #1 – Partition:** Use a hash function h_1 to split tuples into partitions on disk based on target hash key. This will put all tuples that match into the same partition. Assume B buffer pages in total. We will have B-1 output buffer pages for partitions and 1 buffer page for input data. If any partition is full, the DBMS will spill it to disk. Thus, this phase results in $B - 1$ partitions.
- **Phase #2 – ReHash:** For each partition on disk, read its pages into memory and build an in-memory hash table based on a second hash function h_2 (where $h_1 \neq h_2$). This will put all tuples that match into the same bucket. Then go through each bucket of this hash table to bring together matching tuples to compute the aggregation. This assumes that each partition fits in memory.

During the ReHash phase, the DBMS can store pairs of the form (`GroupByKey`→`RunningValue`) to compute the aggregation. The contents of `RunningValue` depends on the aggregation function (e.g. `(COUNT, SUM)` for `AVG`). To insert a new tuple into the hash table:

- If it finds a matching `GroupByKey`, then update the `RunningValue` appropriately.
- Else insert a new (`GroupByKey`→`RunningValue`) pair.

In general for aggregation, hashing is always more efficient unless the data is already sorted beforehand (e.g. following `Order By`).

Lecture #11: Joins Algorithms

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Introduction

The goal of a good database design is to minimize unnecessary information repetition. This is why tables are often composed based on *normalization theory*. Joins are therefore needed to reconstruct the original tables.

We focus on **inner equijoin** algorithms for combining two tables. An *equijoin* algorithm joins tables where keys are equal. These algorithms can be tweaked to support other joins. For binary joins, we often prefer the left table (the "outer table") to be the smaller one of the two.

2 Join Operators

In a query plan, the operators are arranged in a directed tree where the data flows from the leaves towards the root. The output of the root node is the result of the query. The following subsections will discuss the decisions for designing any join operator.

Operator Output

For a tuple $r \in R$ and a tuple $s \in S$ that match on join attributes, the join operator concatenates r and s together into a new output tuple.

In reality, the content of output tuples generated by a join operator varies. It depends on the DBMS's query processing model, storage model, and the query itself. There are multiple approaches to designing the join operator's output.

- **Data:** This approach copies the values for the attributes in the outer and inner tables into tuples put into an intermediate result table just for that operator. The advantage of this approach is that future query plan operators never need to return to the base tables to get more data. This is even more useful in the case of NSMs since all the tuple data will be brought into memory anyway. The disadvantage is that this requires more memory to materialize the entire tuple. This is called *early materialization*.
  The DBMS can also do additional computation and omit attributes that will not be needed later in the query to optimize this approach further.
- **Record Ids:** The DBMS only copies the join keys and the matching tuple record ids in this approach. This approach is ideal for column stores since the join can avoid bringing additional columns into memory to copy them. This is called *late materialization*.

In the real world, DBMS often mix and match between the two approaches, wherein based on the number of tuples, columns, and further operators, a combination of the two strategies can be taken.

Cost Analysis

The cost metric used here to analyze the different join algorithms will be the number of disk I/Os used to compute the join. This includes I/Os incurred by reading data from the disk and writing any intermediate data out to the disk. We seek to reduce I/Os for joins in disk-based systems, ignoring compute costs as I/O costs dominate.

Note that only I/Os from computing the join are considered, while I/O incurred when outputting the result is not. This is because the output cost depends on the data. Moreover, the output for any join algorithm will be the same, and therefore the cost will not change among the different algorithms.

Variables used in this lecture:
- M pages in table R (Outer Table), m tuples total
- N pages in table S (Inner Table), n tuples total

Further, note that $R \bowtie S$ (the natural join of tables R and S) is the most common operation and must be carefully optimized. One inefficient algorithm may involve computing $R \times S$ (the cross product of tables R and S) and selecting the relevant tuples. However, the cross-product is huge and results in a very inefficient approach.

In general, there will be many algorithms/optimizations which can reduce join costs in some cases, but no single algorithm works well in every scenario.

3 Nested Loop Join

At a high level, the join algorithm contains two nested `for` loops that iterate over the tuples in both tables and perform a pairwise comparison. If the tuples match the join predicate, then it outputs them. The table in the outer `for` loop is called the *outer table*, while the table in the inner `for` loop is called the *inner table*.

The DBMS will always want to use the "smaller" table as the outer table. Smaller generally is based on the number of pages but can sometimes mean tuples as well. We will buffer as much of the outer table in memory as possible, and use indexes where available.

Naive Nested Loop Join

Compare each tuple in the outer table with each tuple in the inner table. This is the worst-case scenario where the DBMS must scan the entirety of the inner table for each tuple in the outer table without any caching or access locality.
Cost: $M + (m \times N)$

Block Nested Loop Join

For each block in the outer table, fetch each block from the inner table and compare all the tuples in those two blocks. This algorithm performs fewer disk accesses because the DBMS scans the inner table for every outer table block instead of for every tuple.
Cost: $M + ((\#\text{ of blocks in } R) \times N)$

Block Size: If the DBMS has B buffers available to compute the join, it can use $B - 2$ buffers to scan the outer table. The table with less number of pages will be the outer table. It will use one buffer to scan the inner table and one buffer to store the join's output.
Cost: $M + \left(\left\lceil \frac{M}{B-2} \right\rceil \times N\right)$

Index Nested Loop Join

The previous nested loop join algorithms perform poorly because the DBMS has to do a sequential scan to check for a match in the inner table. However, if the database already has an index for one of the tables on the join key, it can use that to speed up the comparison. The DBMS can use an existing index or build a temporary one for the join operation.

The outer table will be the one without an index. The inner table will be the one with the index.

Assume the cost of each index probe is some constant value C per tuple.
Cost: $M + (m \times C)$

4 Sort-Merge Join

At a high level, a sort-merge join sorts the two tables on their join key(s). The DBMS can use the external mergesort algorithm for this. It then steps through each table with cursors and emits matches (like in Mergesort).

This algorithm is useful if one or both tables are already sorted on join attribute(s) (like with a clustered index) or if the output needs to be sorted on the join key anyway.

The worst-case scenario for this algorithm is if the join attribute for all the tuples in both tables contains the same value, which is very unlikely to happen in real databases. In this case, the cost of merging would be $M \cdot N$ since, for each outer page, we will have to match the entire inner table. Most of the time, though, the keys are mostly unique, so the merge cost can be assumed to be approximately $M + N$.

Assume that the DBMS has B buffers to use for the algorithm:

- Sort Cost for Table R: $2M \times \left(1 + \lceil log_{B-1} \lceil \frac{M}{B} \rceil \rceil\right)$
- Sort Cost for Table S: $2N \times \left(1 + \lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil\right)$
- Merge Cost: $(M + N)$

Total Cost: Sort + Merge

5 Hash Join

Hash join algorithms use a hash table to split the tuples into smaller chunks based on their join attribute(s). This reduces the number of comparisons that the DBMS needs to perform per tuple to compute the join. Hash joins can only be used for equi-joins on the complete join key.

If a tuple $r \in R$ and a tuple $s \in S$ satisfy the join condition, then they have the same value for the join attributes. If that value is hashed to some value i, the R tuple has to be in bucket r_i, and the S tuple has to be in bucket s_i. Thus, the R tuples in bucket r_i need only to be compared with the S tuples in bucket s_i.

Basic Hash Join

- **Phase #1 – Build:** First, scan the outer relation and populate a hash table using the hash function h_1 on the join attributes. The key in the hash table is the join attributes. The value depends on the implementation can be full tuple values or a tuple ID (Early vs. Late Materialization).
- **Phase #2 – Probe:** Scan the inner relation and use the hash function h_1 on each tuple's join attributes to jump to the corresponding location in the hash table and find a matching tuple. Since there may be collisions in the hash table, the DBMS must examine the original values of the join attribute(s) to determine whether tuples satisfy the join condition.

One optimization for the probe phase is the usage of a Bloom Filter. This is a probabilistic data structure

that can fit in CPU caches and answer the question *is key x in the hash table?* with either *definitely no* or *probably yes*. It reduces the amount of disk I/O by preventing disk reads that will not result in a match. Such techniques of providing extra metadata are called **sideways information passing**.

If the DBMS knows the size of the outer table, the join can use a static hash table. If it does not know the size, the join must use a dynamic hash table or allow for overflow pages.

Grace Hash Join / Partitioned Hash Join

When the tables do not fit in the main memory, the DBMS has to swap tables in and out at random, leading to poor performance and not leveraging the buffer pool. The Grace Hash Join is an extension of the basic hash join that also hashes the inner table into partitions that are written out to disk.

- **Phase #1 – Build:** First, scan both the outer and inner tables and populate a hash table using the hash function h_1 on the join attributes. The hash table's buckets are written out to disk as needed. If a single bucket does not fit in memory, the DBMS can use *recursive partitioning* with different hash function h_2 (where $h_1 \neq h_2$) to further divide the bucket. This can continue recursively until the buckets fit into memory.
- **Phase #2 – Probe:** For each bucket level, retrieve the corresponding outer and inner tables pages. Then, perform a nested loop join on the tuples in those two pages. The pages will fit in memory, so this join operation will be fast. (If the number of tuples with the same join key does not fit in a single block, then perform a block nested loop join).

Partitioning Phase Cost: $2 \times (M + N)$
Probe Phase Cost: $(M + N)$
Total Cost: $3 \times (M + N)$

Hybrid hash join optimization: adapts between basic hash join and Grace hash join; if the keys are skewed, keep the hot partition in memory and immediately perform the comparison instead of spilling it to disk. Difficult to implement correctly and rarely done in practice.

6 Conclusion

Joins are an essential part of interacting with relational databases, and it is, therefore critical to ensure that a DBMS has efficient algorithms to execute joins.

Algorithm	I/O Cost	Example
Simple Nested Loop Join	$M + (m \cdot N)$	1.4 hours
Block Nested Loop Join	$M + (M \cdot N)$	50 seconds
Index Nested Loop Join	$M + (m \cdot C)$	Varies
Sort-Merge Join	$M + N +$ (sort cost)	0.75 seconds
Hash Join	$3 \cdot (M + N)$	0.45 seconds

Figure 1: The table above assume the following: $M = 1000, m = 100000, N = 500, n = 40000, B = 100$ and 0.1 ms per I/O. Sort cost is $R + S = 4000 + 2000$ IOs, where $R = 2 \cdot M \cdot \left(1 + \lceil \log_{B-1} \lceil M/B \rceil \rceil\right) = 2000 \cdot (1 + \lceil \log_{99} \lceil 1000/100 \rceil) = 4000$ and $S = 2 \cdot N \cdot \left(1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil\right) = 1000 \cdot (1 + \lceil \log_{99} \lceil 500/100 \rceil) = 2000$.

Hash joins are almost always better than sort-based join algorithms, but there are cases in which sorting-based joins would be preferred. This includes queries on non-uniform data, when the data is already sorted on the join key, or when the result needs to be sorted. Good DBMSs will use either or both.