Storage, Indexing, and Recovery

CMSC 206 – Database Management Systems

Revision History
26 Nov 2022: Examples update – Ria Borromeo
07 November 2020: Main content – Thomas Laurent

Outline

1. Introduction
2. Storage
3. Indexes
4. Indexes in practice

Introduction

In the previous lecture, we saw that the DBMS is somewhat smart and tries to optimize our queries by executing operations in a certain order. We also saw that the way we write our queries can help performance.

In this module, we will see another important tool for performance in DB: indexing.

Storage

Different kinds of storage

The information system has access to different kinds of storage that come with different speeds and that are more or less suited to different operations.

The rough idea is that the larger the storage, the slower it is, as fast storage is expensive.

The different kinds of storage include:

- RAM: very fast, expensive, and relatively small; volatile.
- SSD: faster and more expensive than HDD; non-volatile.
- HDD: slow and relatively cheap; non-volatile.
- Magnetic tape: slowest and largest, usually for archiving purposes; non-volatile.

Cost of operations

- No matter what the storage method is and no matter how the storage is physically organized (e.g. heap file, sorted file, hashed file, …), reading and writing always has a (time) cost. Thus we want to minimize the operations needed by our DB.
- Indexes will help minimize the read operations necessary when searching for data, i.e. running queries.

Indexes

Different kinds of indexes

- There are different kinds of indexes that suit different needs.
- We can build indexes on a single column or on multiple columns.
- We can use different types of indexes: single- or multi-level, hash index (good for equality evaluation) or tree index (good for range queries)…

The other side of the coin

- Indexes seem perfect if they speed our queries! So why not just build indexes on every column in our database then have the fastest DB ever?
- Indexes do speed up query execution (most times, for some types of queries an index would not help) but they also have a cost:
  - An index contains data, which means it must be stored.
  - An index must be up-to-date to be of any use, which means we need to update it every time we insert or delete some records in the DB. This means that although we may be speeding up our queries, we are slowing down other operations.

Indexes in practice

Default indexes

- Actually, you have been using indexes without knowing it.
- SQLite and most DBMS, create an index by default on the primary key columns, as the primary key is often used to search for records or to join tables.
- You can see this index for yourself in SQLite by typing:

```
.indexes
```

Making indexes

Syntax

```
CREATE INDEX index_name ON table_name (column_names)
DROP INDEX index_name
```

We create indexes with the CREATE INDEX command. The basic usage only needs the table name and the column names to build the index on. Each DBMS then lets us specify options for the index, such as the index method (hashing, tree, …) that we want to use or the order of the index for example.

We can get rid of an index when it is not needed anymore by using the `DROP INDEX` command.

When to use indexes?

- Some rules of thumb when we might need an index:
  - A column is often used to select a record (a column is always used in the where clause)
  - A column is often used to join tables (it is always used in a join clause)
  - A table is big enough (if I have two records in my table, reading my index is as time consuming as reading my table…)

Recovery

Introduction

- Recovery is how the DBMS copes when a failure happens. It is strongly tied with ACID properties.
- There are many different ways recovery mechanisms can be implemented in a DBMS. As usual, how things work will depend on your DBMS and how it is configured. Your DBMS's documentation should always your main source of information.
- In these slides, we will introduce the main concepts linked with DB recovery and give a very high level view of what goes on under the hood.

Crashes

Crashes

- Crashes can happen for all kinds of reasons, even with the best thought out well and implemented system, e.g.:
  - Physical problems: power outage, hardware failure
  - Software error: division by 0, buffer overflow
  - Concurrency deadlock

- Our system should be able to recover from these crashes. Hopefully rapidly and with no (or minimal) data loss.

Crashes and ACID properties

- **Durability** is the property that is most obviously linked to crashes: once a transaction is committed, it should stay committed, no matter what, i.e. we should not lose data even if there were a crash.
- **Atomicity** and **consistency** are also linked with crashes: if a crash happens in the middle of a transaction, we want to make sure the whole transaction is either committed or cancelled and the database is kept in a consistent state after the crash.
- In the case of a soft/local crash (e.g. error while executing the transaction), we just rollback. However, for hard/system crashes, things are more complicated.

Backups and logs

Backups

- Backups are a snapshot of the database (or of a system in general) in a consistent state at a past point in time.
- Dump and restore tools are available for most database management systems to create backups and restore the database to the state represented by a backup.
- REMINDER: Test your backups
  - Making backups without verifying that you can restore them just leaves you with a Schrödinger's backup: maybe you can restore your DB and recover from a crash, maybe not

Logs

- The log journal keeps a trace of all the operations that happen on the DB. It can be used to replay these transactions at a later date if we need to perform operations whose effect was lost after a crash.
- Example:

  Start transaction T
  Write X
  Write Y
  Delete Z
  Commit T

- The system regularly makes synchronization points where all operations until that point are committed and written to disk so we know that the system is in a consistent state.

Recovery

Media (disk) crashes VS other crashes

- If the disk was damaged in some way in the crash and we lost data from the database then we need to restore a backup.
- We can replay the committed operations in the logs from the time of the backup up to the time of the crash to get back to the situation before the crash.
- If no data was lost during the crash then we do not necessarily need to restore a backup and we can recover using the logs.

Replaying the logs

- The idea here is we want to replay (redo) committed operations that happened before the crash but were not yet written to disk, OR undo operations that were written to disk but were not committed at the time of the crash. There are two main strategies for this:
  - NO-UNDO/REDO
  - UNDO/REDO

NO-UNDO/REDO

- This strategy is used in *deferred update* systems, where changes are written to disk only after they are committed. When a crash happens, we know that all the data on disk corresponds to committed operations so we do not need to undo any operations.
- There might be operations that were committed but not yet written to disk at the time of the crash though. In this case we need to redo these operations.

UNDO/REDO

- This strategy is used in *immediate update* systems where changes can be written to disk before a transaction is committed. In this case we might have changes committed to disk when the crash happens but that were not yet committed, as well as committed changes that were not yet written to disk. We then need to undo the non-committed changes and redo the committed ones.