



Indexes and Transactions

GRA 4142 Data Management and Python Programming

Jan Kudlicka (jan.kudlicka@bi.no)

Department of Data Science and Analytics

November 12, 2021

```
1 CREATE TABLE person(  
2   id INT NOT NULL PRIMARY KEY,  
3   fname VARCHAR(255) NOT NULL,  
4   lname VARCHAR(255) NOT NULL,  
5   phone CHAR(8)  
6 );
```

SQLite3 database, 10 000 000 people. The file has 475 MB.

- ▶ How long does it take to find name and phone number for person with ID 3,857,845?
- ▶ How long does it take to find the phone number for Øivind Elvik?

```
sqlite> SELECT * FROM person WHERE id=3857845;
id          fname      lname      phone
-----
3857845     Heidi       Waaler     34773357
Run Time: real 0.000 user 0.000100 sys 0.000085
```

```
sqlite> SELECT * FROM person WHERE lname='Elvik' AND fname='Øivind';
id          fname      lname      phone
-----
3323374     Øivind     Elvik      55350459
Run Time: real 0.935 user 0.827037 sys 0.106191
```

0.935 sec is very slow. Why is it so slow and what can we do about it?

The slow query from the experiment performs a **full table scan**: each row in the table is read and checked if it the condition is met.

To see the execution plan (how the query is executed):

- ▶ EXPLAIN QUERY PLAN **SELECT** ... in SQLite
- ▶ EXPLAIN **SELECT** ... in MySQL

The execution plan of the slow query from the experiment:

```
sqlite> EXPLAIN QUERY PLAN SELECT * FROM person WHERE lname='Elvik' AND fname='Øivind';  
QUERY PLAN  
`--SCAN TABLE person
```

Index – a persistent data structure which can be used to quickly locate rows in a table based on the values of one or several given attributes.

Similar idea to an index in a book – keywords are ordered alphabetically and for each keyword there is a list of pages.

Cost of an index:

- ▶ Extra storage space (usually not a problem)

In our experiment, 475 MB before creating an index for the combination of lname, fname, and 692 MB afterwards.

- ▶ Index creation (might take time, but it is done only once)
- ▶ Index maintenance – index must be updated when the data in the table changes (might be a problem)

```
CREATE INDEX index_name ON table(column,column,...);  
CREATE UNIQUE INDEX index_name ON table(column,column,...);
```

A unique index does not allow duplicate values.

Indexes on the primary key attributes are created automatically. Some RDBMS (e.g. MySQL) create indexes on the foreign key attributes automatically as well.

Different SQL syntax for dropping an index in different RDBMS:

SQLite:

```
DROP INDEX table.index_name;
```

MySQL:

```
ALTER TABLE table DROP INDEX index_name;
```

For other systems, check the reference manual.

```
sqlite> CREATE INDEX lname_fname ON person(lname, fname);
```

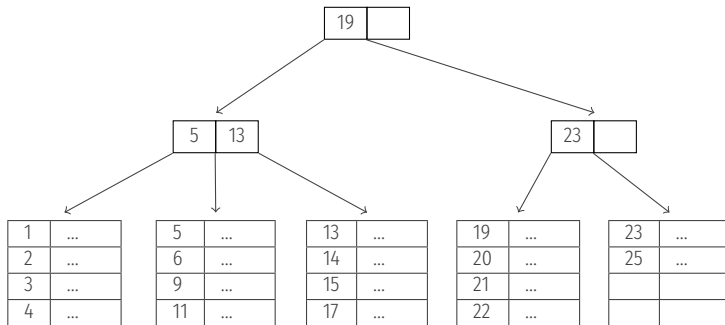
```
sqlite> SELECT * FROM person WHERE lname='Elvik' AND fname='Øivind';
```

id	fname	lname	phone
3323374	Øivind	Elvik	55350459

Run Time: real 0.000 user 0.000142 sys 0.000125

```
sqlite> EXPLAIN QUERY PLAN SELECT * FROM person WHERE lname='Elvik' AND fname='Øivind';  
QUERY PLAN  
`--SEARCH TABLE person USING INDEX lname_fname (lname=? AND fname=?)
```

Using B and B+-trees:



Great explanation of clustered and non-clustered indexes and how they work: <https://youtube.com/watch?v=ITcOiLSfVJQ>

- ▶ Which queries and insert/update/delete statements will be executed?
- ▶ How often?
- ▶ What are the time constraints?
- ▶ How many rows are in the relevant tables?

Tip: Add indexes which might be helpful and use

- ▶ `EXPLAIN QUERY PLAN SELECT ...` in SQLite
- ▶ `EXPLAIN SELECT ...` in MySQL

to see how the queries are executed, what indexes are used and to check if the performance is satisfactory. Then drop the indexes which are not used.

- ▶ Denormalization

$E(\underline{A_1}, A_2), F(\underline{B_1}, B_2), EF(\underline{A_1}, \underline{B_1}) \rightarrow EF(\underline{A_1}, \underline{B_1}, A_2, B_2)$

- ▶ Storing derived attributes

- ▶ Storing summary statistics

Example: Storing the number of employees for each department, storing the sum of hours_spent for each employee.

- ▶ Vertical partitioning

$E(\underline{A_1}, A_2, A_3) \rightarrow E_1(\underline{A_1}, A_2), E_2(\underline{A_1}, A_3)$. E_1 and E_2 might be stored on separate physical location or database servers.

- ▶ Horizontal partitioning (sharding)

Rows are divided into shards which might be stored on separate physical locations or database servers. E. g. logs are or records are partitioned based on the year when the log or record was created.

- ▶ Hardware tuning

```
CREATE INDEX emp_lname ON employee(last_name);
```

```
SELECT *  
FROM employee  
WHERE last_name LIKE '%g'
```

Is the index emp_lname going to be used when executing this query?

1. Yes
2. No

Consider a relation $R(\underline{A}, B, C, D)$ containing 10^7 records. B contains 10^5 distinct values. The following SQL prepared statement is executed very frequently:

```
SELECT C FROM R WHERE B=?
```

Which indexes would you create?

1. One index on B and one on C
2. Index on C
3. Index on B
4. Index on (B, C)
5. Index on (C, B)

Consider a relation $R(\underline{A}, B, C, D)$ containing 10^7 records. A is the primary key, and B contains 10^5 distinct values. The following SQL prepared statements are executed very frequently:

```
UPDATE R SET D=? WHERE B=?  
SELECT D FROM R WHERE C=?
```

Which indexes would you create?

1. One index on B and one on C
2. Index on B
3. One index on D and one on B
4. Index on (C, B)
5. Index on (B, D)

Initial state of the Account table:

AccountNo	Name	Balance
12345678	Jan's checking account	3500
23456789	Jan's savings account	13000
...	...	...

Jan transfers 3000 NOK from his checking account to his savings account:

AccountNo	Name	Balance
12345678	Jan's checking account	500
23456789	Jan's savings account	16000
...	...	...

Write SQLs to execute this money transfer and discuss what can go wrong during their execution.

```
UPDATE Account  
SET Balance=Balance-3000  
WHERE AccountNo=12345678
```

```
UPDATE Account  
SET Balance=Balance+3000  
WHERE AccountNo=23456789
```

- ▶ The first SQL gets executed, followed by a software, hardware or network failure.
- ▶ Both SQLs get executed, but the hardware failure occurs before the changes are permanently stored on a storage device.

Transaction – a series of operations that need to be executed as a single unit of work and that transforms the database from one consistent (and valid) state to another.

Note that the integrity constraints might be violated during a transaction but not when it has finished.

Failure (or a user's decision to abort the transaction) must be treated as if the transaction never happened.

- ▶ **Commit:** making the “effects of the transaction” permanent (successful executing the transaction)
- ▶ **Rollback:** aborting the transaction (because of a failure or user's decision)

Jan has a subscription from Netflix which withdraws 100 NOK each month from Jan's checking account.

What can go wrong if this transfer happens to be executed at the same time as the transfer of money to Jan's savings account?

What are the possible balances at Jan's checking account after executing both transfers?

One of possible schedules:

Transfer 1	Transfer 2
READ <i>balance</i> of 12345678	
$balance \leftarrow balance - 3000$	READ <i>balance</i> of 12345678
WRITE <i>balance</i>	$balance \leftarrow balance - 100$
...	WRITE <i>balance</i>
	...

The account balance after both transfers will be 3400.

Other schedules might lead to the balance 500 and 400.

Databases are usually accessed (both queried and modified) by many clients at the same time.

To avoid problems with concurrent (parallel) execution (such as the example on the previous slide) a transaction schedule must be **serializable**:

Execution of concurrent transactions must be equivalent to a serial execution of the transactions.

Example: In our example, the execution must be equivalent either to transferring money to Jan's savings account first and then to Netflix, or to Netflix first and then to Jan's savings account.

Note that in general different serial orders might lead to different results. If we care about the order we must run the transactions serially (rather than running them concurrently).

ACID—a set of required properties of transactions:

- ▶ **Atomicity:** all statements in a transaction are executed or none.
- ▶ **Consistency:** all database constraints are satisfied after a transaction is executed.
- ▶ **Isolation:** the result of executing concurrent transactions is the same as if the transactions were executed serially.
- ▶ **Durability:** once a transaction finishes, effects of the transaction are permanently stored in the database.

- ▶ To start a transaction:

```
START TRANSACTION
```

or

```
BEGIN
```

- ▶ To commit the transaction (making the changes permanent):

```
COMMIT
```

- ▶ To roll back the transaction (canceling the changes):

```
ROLLBACK
```

```
1 BEGIN;  
2  
3 UPDATE Account  
4 SET Balance=Balance-3000  
5 WHERE AccountNo=12345678;  
6  
7 UPDATE Account  
8 SET Balance=Balance+3000  
9 WHERE AccountNo=23456789;  
10  
11 COMMIT;
```

The ACID isolation property is often relaxed to reduce the overhead and increase the concurrency. Three weaker isolation levels that allow some serializability violations in order to achieve higher performance.

Isolation level is set per transaction and it reflects what *read phenomena* might occur in the current transaction:

- ▶ **Dirty reads:** reading data written by another transaction before it is committed
- ▶ **Non-repeatable reads:** retrieving the same row twice might return different data (due to changes by other committed transactions)
- ▶ **Phantom reads:** new rows might appear in tables during the transaction (added by other committed transactions)

Note that the description of these reading phenomena is somewhat simplified.

Level	Dirty reads	Non-repeatable reads	Phantom reads
READ UNCOMMITTED	May occur	May occur	May occur
READ COMMITTED	—	May occur	May occur
REPEATABLE READ	—	—	May occur
SERIALIZABLE	—	—	—

SQL to change the isolation level (of the next transaction):

```
1 SET TRANSACTION ISOLATION LEVEL level
```