Project 1: Index performance experiments

T-508-GAG2 Performance of Database Systems

Instructors: Björn Þór Jónsson, Grímur Tómas Tómasson

Teaching assistant: Patrekur Patreksson

March 27th 2019

Group name: Unicorns

Names:

Daníel Örn Melstað (danielm15@ru.is)

Halldór Bjarni Stefánsson (halldors17@ru.is)

Hera Hjaltadóttir (hera16@ru.is)

Martha Guðrún Bjarnadóttir (marthab16@ru.is)

Víðir Snær Svanbjörnsson (vidir17@ru.is)

Table of Contents

Abstract

Understanding the impact of different index implementations on query processing performance is useful to let database management systems (DBMS) perform better. Experimenting with different indexes and analysing the results is a good way to see their usefulness. This report describes such an experiment, aiming to determine the strengths and weaknesses of different indexes on certain hardware. Nonclustered indexes can show faster performance with small fractions of records retrieved but performance reduces with larger fractions. Clustered indexes are fast when working with a big range of information retrieved and no indexes worsen with more data.

1. Introduction

Looking up specific records from a database management system with Structured Query Language (SQL) can be done by scanning the entire file record, but if the database is very large the performance can be greatly affected. That's why databases should be optimized so that records can be retrieved faster. Because of today's demand for getting data fast, lacking query optimization can discourage usage of certain software [5].

An optimization technique called *indexing* can help when accessing records in different ways to retrieve what is being asked for [1]. An *index* is a structure associated with retrieving records and data from a database in a faster way than the typical scanning method. On SQL servers, the index keys are stored in a B^+ tree [3]. Clustered indexes are sorted and store their key values in a sorted manner like the data rows they are connected to [1]. The data pages are stored in the leaf nodes of the index [10]. Nonclustered indexes have a different structure than the data rows. These indexes store pointers that point to each record [1]. The indexes do not sort the rows physically and the data pages are not stored in the tree [10]. When SQL Server scans a sorted database with no index, it locates the position of a data record using a heap [4].

The following experiment shows the result of different index implementations on query processing performance and benchmarks the hardware used for SQL server. The SQL script used was produced by a script coded in Python and then executed in Microsoft SQL Server 2017.
Chapter 2 will go into the background of indexes, where we discuss the difference between their implementations in Ramakrishna’s book and Microsoft SQL Server. Chapter 3 describes the technical environment used and the database provided, along with which measurements were made and how they were performed. Chapter 4 describes the results of our measurements, followed up with a discussion in chapter 5 and a short conclusion in chapter 6.

2. Background

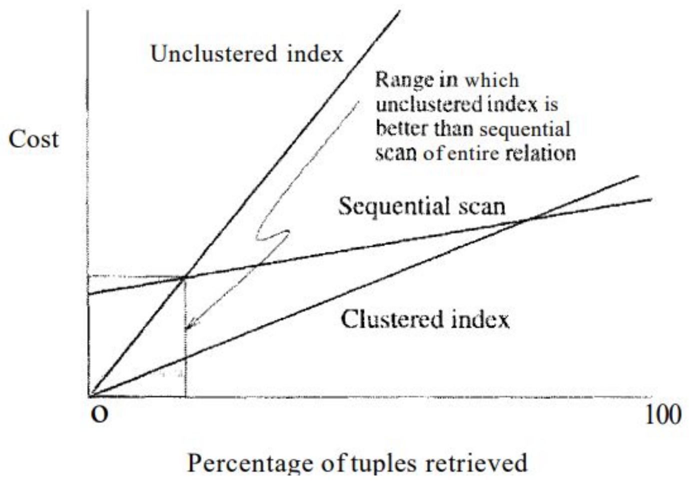

Figure 1: Figure 20.2 [8], showing the impact of clustering.

A book for this course has a graph showing the performance difference between using clustered, nonclustered and no index, shown above in Figure 1. The experiment described in this report is a similar repetition of the experiment in the book, but while the experimenters in it probably used an older HDD, in this experiment the same was done except with today's hardware, an SSD.

2.1 Indexes in Ramakrishnan's book

As described in the book, an index is the addition of a collection of data entries onto a database. The collection is much smaller than the database and can therefore locate data entries much faster than we would otherwise [1].

There are a few ways described to organize the internal structure of an index. One of the methods mentioned is using a *clustered index*. With a clustered index, the data records in a database are sorted in the same or a similar manner as they are in the index. When an index doesn't have this property, it is *nonclustered*. Using a clustered index has its downside, since inserts to the database will take extra time to keep the data somewhat sorted. The main focus of this experiment was to find out whether this extra time cost makes nonclustered indexes a more feasible option [1].

To use as little extra time as possible for operations, the book comes up with two efficient ways to easily keep track of the entries in your index. The first method mentioned is an *Indexed Sequential Access Method* (ISAM). When using ISAM, the index stores sequentially the first key on each page in the database along with a pointer to the key's location. The second method is storing the contents of an index in a *B^+-tree*, which is a self-balancing tree search structure that maintains the semi-sorted order of the data. The leaves of the tree contain pointers to records in the table. Although each method has its advantages and disadvantages, B^+-trees generally perform better than ISAM [3].

2.2 Indexes in SQL Servers

In SQL servers, indexes behave in the same way as described in Ramakrishnan's book. The contents of an index are kept in a B^+-tree, which is also the book's preferred method of storing entries. On tables with no index, the book describes that the server performs a full-table scan each time they scan for a certain value. This however does not describe reality, since in real life applications the server will store pages in a heap and scan the heap for a value [4].

3. Methods

All experiments were performed on a single laptop computer using a given database for the experiment. The database contained one million records and each one was a little over 200 bytes. The different data points to use for the measurements, to retrieve specific fractions of the database had to be decided by the group.

3.1 Hardware

The experiment was performed on a Lenovo Yoga 720 laptop.
The computer's CPU, SSD and RAM for the experiment is the following:

CPU: SSD: NVMe Intel SSDPEKKW25

RAM: DDR4 8GB 2133MHz

Storage: INTEL SSD MODEL SSDPEKKW256G7 with 237 GB storage.

3.2 Software environment

Server:	Microsoft SQL Server Enterprise 2017
Program:	Microsoft SQL Server Management Studio
OS:	64-bit Windows 10

3.3 Database

The database used in the experiment, *Project1*, was provided by the instructor for the course. It is made up of one table called Wisc1000K, containing exactly one million records. Each record is about 200 bytes, therefore the database is around 200 MB in total.
The table consists of 26 columns, three of them were used for this experiment: UNIQUE1, UNIQUE2 and UNIQUE3. The Database has a clustered index on UNIQUE2, nonclustered index on UNIQUE1 while UNIQUE3 is not indexed.

3.4 Queries

For the experiment we executed hundreds of SQL queries. Each query simply counts the number of rows in our selection by running '`COUNT`' on the UNIQUE3 column. We chose the UNIQUE3 column since it has no index, to avoid the measurements being affected by anything else but the index type we are testing each time. The reason we are using '`COUNT`' is to avoid the increase in running time that comes with writing out each row of the selection. Instead of multiple rows we get only one row containing the number of rows in the selection, which drastically improves the running time especially when we are measuring large fractions of the database.

For each of the three index types (clustered, nonclustered and no index) we measured the performance when selecting 23 different fractions, the first 12 of them increasing in very small increments from 0.0001% to 5%. From 5% to 30% we increase them in 5% increments, then after that in 10% increments, until we are selecting 100% of the database. To select the fractions we used '`WHERE {column in index} < {fraction * no. rows in table}`', which works because the values in the columns are sequential.

We fetched each fraction five times, measuring the time taken for each run. After each run we cleared buffers using '`DBCC DROPCLEANBUFFERS`' and '`DBCC FREEPROCCACHE`'. Each measurement was then documented in an SQL table. Afterwards we removed the longest and the shortest time of each set of five measurements, before calculating the average of the other three measurements.

In total we ran 3 * 23 * 5 = 345 SQL queries (3 index types, 23 fraction values, 5 runs for each index-fraction combination). Knowing this beforehand, we decided against handwriting each query and wrote a short Python script to generate the queries. An example query along with explanations is shown in appendix.

4. Results

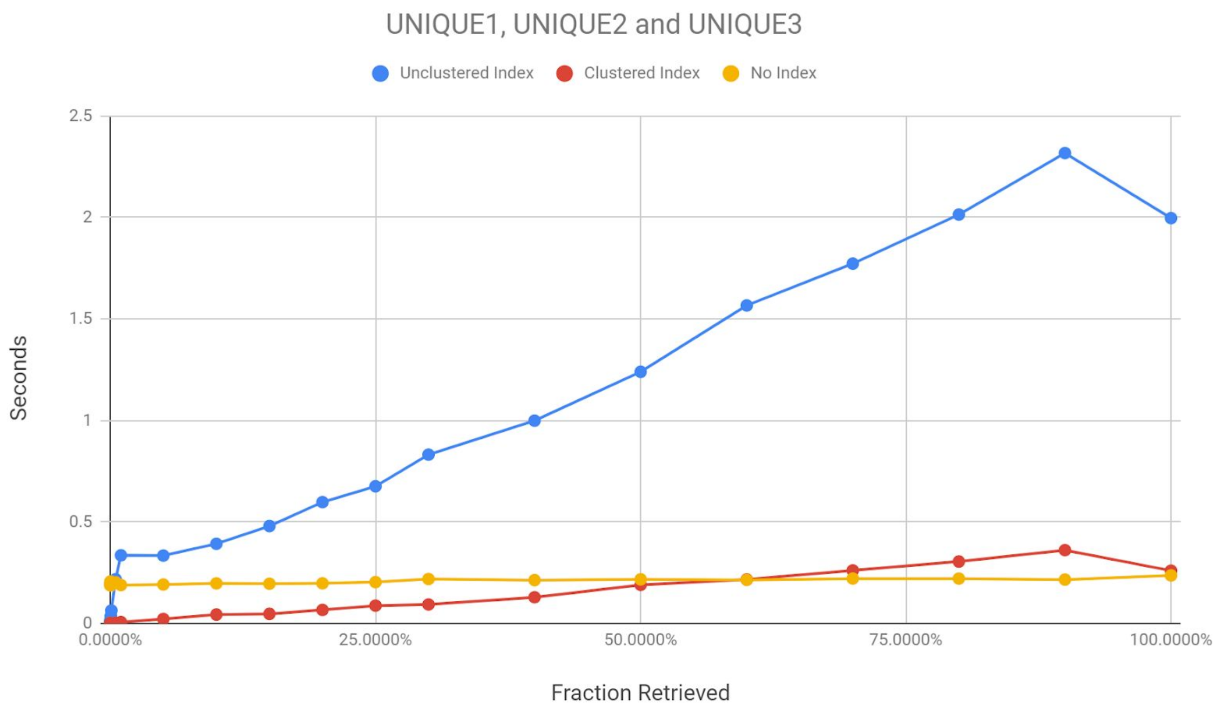

Figure 2: Impact of indexes on time taken to retrieve data

Examining these three variables in figure 2, the most efficient one is the clustered index and the nonclustered index is least efficient, except when retrieving less than 0.5% of the data. The figure shows how long it took (in seconds) to retrieve various fractions of the data using each of the columns.

4.1 Nonclustered Index

By taking a closer look at the query execution plan for the nonclustered index in Microsoft SQL Server Management Studio for 0.1% and 5%, we noticed a difference in the query plan, which is why the plot goes from such a steep upward slope to a more horizontal behavior, the only difference we found in the query execution plan is that the SQL server uses parallelism when it has reached about 5%. Parallelism is used implicitly to speed up the queries [9]. At around 50% on the nonclustered index the optimizer throws in a sort after the nested loop (inner join) and stays all the way to 100%. When trying to explain the downward slope of the same index from 90% to 100% we found no difference in the execution plan. However we had some theories on why this downward slope was happening that will be explored further in the discussions section below.

4.2 Clustered Index

When looking at the clustered index and its performance the optimizer adds parallelism and an extra stream aggregation after the fraction retrieved has reached about 20% and keeps that query plan all the way to 100%. It gets slightly slower as the fraction retrieved gets larger, but overall performs orders of magnitudes faster than the nonclustered index. Here we also saw the same downward slope from 90% to 100% as we saw for the nonclustered index, although less dramatic.

4.3 No Index

The query plan for the no index stays the same no matter the fraction retrieved, 99% of the cost of retrieving data with no index is in the heap scan. The execution time also stayed the same for each fraction retrieved using the column with no index. This is caused by the SQL server using read ahead operations in order to minimize the total number of read operations, which causes it to scan all the pages in the heap sequentially [11].

5. Discussions

The results are similar to a Ramakrishnan's book used in the course except that the nonclustered index and the clustered index times doesn't go completely linear in our experiment. This might be caused by the query optimizer behaving differently. Those results possibly come from an experiment using a HDD but our measurements were made with a SSD. Other reasons could be that the architecture of SQL server could be different than the one used in the book and the queries used for measurements are different [8].

Although we couldn’t see a difference in the execution plan for the SQL server when it fetched 90% of the data versus 100% in the data, our group had a theory that the SQL server stores the maximum value of our table and recognized that our query was requesting the whole table. This could possibly explain why it took a slightly shorter time both for the clustered and nonclustered method to fetch 100% of the data than 90%, as it would not need to spend any time scanning and comparing the values. Upon closer inspection of the table, we noticed that SQL server stores statistics for the columns we used. We created a new table identical to the given one, but with no stored statistics, and then we executed the same queries as before on this new table. Now fetching 100% of the data took longer than 90%, confirming our theory.

The evolution of hard disks has been rapidly increasing their performance and storage going from storing 3.75 MB in 1956 [6] up to size of 15 TB HDD in 2018 [7] that can store 4 million times more data. Today most of us seem to have solid state drives (SSD) in our computers that are much faster than HDDs, but though the storage drives keep getting faster, the amount of data is growing and optimizing a DBMS could save money because of expensive hardware, make customers happier, let companies be better prepared for the future of big data, and so on.

6. Conclusions

When thinking about whether to use clustered or nonclustered indexes when designing a database, factors like selectivities and frequency of the queries have to be taken into account [8]. So when choosing clustered or nonclustered indexes in a database, the best course of action depends on how the database is used. Nonclustered can be good to retrieve small amount of information like one specific name of a person but clustered is better if queries retrieving information within a given range are frequent.

References

[1] Raghu Ramakrishnan and Johannes Gehrke, “Overview of storage and indexing” in *Database Management Systems, 3rd ed.* New York, USA: McGraw-Hill, 2003, ch 8.

[2] Raghu Ramakrishnan and Johannes Gehrke, “Storing data: disk and files” in *Database Management Systems, 3rd ed.* New York, USA: McGraw-Hill, 2003, ch 9.

[3] Raghu Ramakrishnan and Johannes Gehrke, “Tree-structured indexing” in *Database Management Systems, 3rd ed.* New York, USA: McGraw-Hill, 2003, ch 10.

[4] Contributors. “Clustered and Nonclustered Indexes Described.” Microsoft. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-2017 (accessed Jan. 28th, 2019).

[5] Karthik Ramachandra. “Optimizing imperative functions in relational databases with Froid.” Microsoft Research blog. https://www.microsoft.com/en-us/research/blog/optimizing-imperative-functions-in-relational-databases-with-froid/ (accessed Jan. 28th, 2019).

[6] Rich Schwerin. “Time Capsule.” Oracle Magazine. https://blogs.oracle.com/oraclemagazine/time-capsule-v7 (accessed Jan. 29th, 2019).

[7] Western Digital. “Ultrastar DC HC600 SMR Series Purpose-Built Efficiency for Sequential Write Workloads”. Western Digital Products. https://www.westerndigital.com/products/data-center-drives/ultrastar-dc-hc600-series-hdd (accessed Jan. 29th, 2019).

[8] Raghu Ramakrishnan and Johannes Gehrke, “Physical database design and tuning” in *Database Management Systems, 3rd ed.* New York, USA: McGraw-Hill, 2003, ch 20.

[9] Paul White, “Understanding and Using Parallelism in SQL Server.” Redgate Hub. https://www.red-gate.com/simple-talk/sql/learn-sql-server/understanding-and-using-parallelism-in-sql-server/ (accessed Jan. 30th, 2019).

[10] Mohammad Sajid. “Know when to use nonclustered index in SQL server database.” StepUp Analytics. https://stepupanalytics.com/when-to-use-non-clustered-index-in-sql-server/?fbclid=IwAR04BQAUKylE90LEvtmE11F5JQ7RuDBMEX43wmwN4WJouxcVivy4PSWgRp8 (accessed Jan. 30th, 2019).

[11] Craig Guyer et al., “Heaps (Tables without Clustered Indexes)” Microsoft SQL Docs https://docs.microsoft.com/en-us/sql/relational-databases/indexes/heaps-tables-without-clustered-indexes?view=sql-server-2017 (accessed March 27th, 2019).

Appendix

An example query:

```
--Here we start the timer for the query:
set @startdate = getdate()

--Here we run the query, forcing nonclustered index:
select count(UNIQUE3)
from Wisc1000K with (index(IX_Wisc1000Ka))
where UNIQUE1 < 100000

--Here we calculate the time taken to run the query:
set @enddate = getdate()
set @time = (convert(numeric(15,8),@enddate)
-convert(numeric(15,8),@startdate)) * (24.0*60.0*60.0)

--Here we insert the time measurement into a table for
later
--analysis:
insert into Measurements (ind, n, el_time)
values ('UNIQUE1', 100000, @time);
```

```
--Here we clear the buffer for accurate performance:
DBCC DROPCLEANBUFFERS
DBCC FREEPROCCACHE
```