CS 587: DBMS Benchmarking Part 2

# DBMS Choice

I will benchmark Microsoft SQL Server 2017 on Windows 10 Pro against PostgreSQL 11 on Linux (tentatively Fedora 30), both running natively, using a solid-state drive for the OS and server and the same magnetic hard drive to host the databases.

I chose SQL Server as my first DBMS because my classmates with industry experience in databases have repeatedly and vocally espoused the merits of Microsoft’s database offering and of the team behind it.

My experience with DBMSes thus far has involved small-scale Web projects running MySQL and SQLite, introductory classes taught using Oracle Database and Postgres, and a database-driven research project running Postgres with over 50 million rows, with individual tables of up to 24 million rows. Of these DBMSes, I have found that Postgres is my personal favorite. Among the same classmates who favor SQL Server, Postgres is also the universal top choice for an unpaid DBMS and a solid second choice for relational databases in general.

Comparing SQL Server with Postgres will allow me to quantify and illuminate the advantages of SQL Server over Postgres that my classmates have so boldly claimed, helping me to form my own opinion and preference between the two. It will also allow me to compare DBMS performance on Windows with NTFS, where SQL Server is the premier choice (as far as I am aware), with that of Linux with ext4, the native and preferred environment for Postgres, which will help inform my future choices as well. Finally, since I am already familiar with Postgres, this comparison will allow me to bridge the divide between the two systems and familiarize myself with SQL Server in what I hope will be an approachable way.

# System Research

## Indexes

**SQL Server**

SQL Server supports a huge range of indexes and index optimizations for different use cases. [[1]](#footnote-1) In SQL Server, there are at least four physical data formats[[2]](#footnote-2):

* Columnstore indexes store data in a columnar format.
* Clustered indexes store data as a B-tree.
* Memory-optimized tables store data as a collection of versioned rows that can be efficiently accessed in-memory, which “is used to allow concurrent reads and writes on the same row.”[[3]](#footnote-3)
* Heaps store data without any inherent organization and is the default if none of the above index types are used.

To clarify: yes, columnstore indexes and clustered indexes are both index formats and physical storage formats, and using either index type causes SQL Server to store the tables accordingly.

The indexes SQL Server supports are as follows:1

**In-memory indexes:**

These indexes are never written to disk and are optimized for in-memory access. For instance, unlike on-disk indexes, in-memory indexes are not divided into pages.

* In-memory hash index
* In-memory, nonclustered B-tree index

**On-disk indexes:**

* Clustered B-tree index
* Nonclustered B-tree index
* Nonclustered index with included columns, “A nonclustered index that is extended to include nonkey columns in addition to the key columns.”1
* Index on computed columns, “An index on a column that is derived from the value of one or more other columns, or certain deterministic inputs.”1
* Filtered nonclustered index, which includes only a specified subset of rows

**Miscellaneous indexes:**

* Unique index, which is functionally identical to a UNIQUE constraint and used to enforce it
* Spatial index, which handles the **geometry** data type efficiently for some operations
* XML index, which handles the **xml** data type efficiently, allowing for XML queries within an XML column
* Full-text index, which uses the Microsoft Full-Text Engine for SQL Server; “it provides efficient support for sophisticated word searches in character string data.”1

**Postgres**

Postgres does not support clustered indexes of any type. Instead, it supports a one-time CLUSTER operation that “instructs PostgreSQL to cluster the table specified by table\_name based on the index specified by index\_name. … When a table is clustered, it is physically reordered based on the index information. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order.”[[4]](#footnote-4) Postgres supports the following index types:[[5]](#footnote-5)

1. B-tree index, which is the default
2. Hash index
3. GiST (generalized search tree) index, which is “an infrastructure within which many different indexing strategies can be implemented.” GiST can be used to define indexes for custom data types.[[6]](#footnote-6)
4. SP-GiST (space-partitioned generalized search tree) index, which can index spatial data, searching through space by partitioning it.[[7]](#footnote-7)
5. GIN (generalized inverted) index, “which [is] appropriate for data values that contain multiple component values, such as arrays. An inverted index contains a separate entry for each component value, and can efficiently handle queries that test for the presence of specific component values.”5
6. BRIN (block range index), which “store[s] summaries about the values stored in consecutive physical block ranges of a table.”5

## Join algorithms

**SQL Server**

According to Microsoft, SQL Server supports merge, hash (in-memory, grace, and recursive), and nested loop joins, but the company appears to guard the details of its join optimizations closely. The official documentation simply states: “When SQL Server processes joins, the query engine chooses the most efficient method (out of several possibilities) of processing the join. The physical execution of various joins can use many different optimizations and therefore cannot be reliably predicted.”[[8]](#footnote-8) I conjecture that SQL Server may rely on its extremely powerful indexes to provide efficient join operations.

Note: in-memory, grace, and recursive hash joins are standard terms that SQL Server and Postgres share. Grace appears to be the three-stage hash join we discuss in class; recursive appears to be an extension of this notion to arbitrary levels of recursion (as memory requirements may dictate).

**Postgres**

I have been unable to find anywhere that Postgres explicitly lists its join algorithms, perhaps because sort-merge, hash, and nested loop are the three, standard, well-known types of join algorithms (in addition, of course, to index nested-loop joins). However, the Postgres query planner’s documentation exposes configuration flags for merge, hash, and nested loop joins. In addition, it allows parallel hashing and partitionwise joins.[[9]](#footnote-9) Partitionwise joins are specialized equi-joins that work by partitioning the join values, optimizing the partitions (e.g. by pruning ones that will produce no results), and joining each partition.[[10]](#footnote-10)

## Buffer Pool

**SQL Server**

SQL Server dynamically allocates and frees memory as needed; it does not use a fixed-size memory allocation or buffer pool. The buffer pool will respect the minimum and maximum memory settings, but as long as the server’s usage is below the maximum and the OS indicates that there is still plenty of free physical memory, the buffer pool will not free any of its pages. I have found no information on the structure of the buffer pool. The following settings control SQL Server’s memory usage:[[11]](#footnote-11)

min\_server\_memory and max\_server\_memory – handle “the SQL Server memory allocation, compile memory, all caches (including the buffer pool), query execution memory grants, [and] lock manager memory.”

memory\_to\_reserve – covers CLR allocations, thread stacks memory, and direct allocations from Windows.

By default, the minimum and maximum server memory are set to allow SQL server to use effectively zero to unlimited memory,[[12]](#footnote-12) and memory to reserve is set to 256 MB.11

SQL Server also supports buffer pool extension, which utilizes a solid-state drive to expand the buffer pool by 1 to 32 times.[[13]](#footnote-13) Based on my server’s settings, this feature is disabled by default.

**Postgres**

The Postgres manual does not address the structure of the buffer pool. However, it does provide the following, crucial controls for memory management:

shared\_buffers controls the size of the shared memory pool (i.e. the buffer pool). It starts out at 128MB but should be set to between 4 GB and 6.4 GB for the test system (which has 16 GB of RAM), per the manual’s recommendation: “If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for shared\_buffers is 25% of the memory in your system. … because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to shared\_buffers will work better than a smaller amount.” [[14]](#footnote-14)

work\_mem controls the amount of memory that in-memory operations like sorts and hashes can use. It starts out at 4 MB, and the manual does not provide any specific tuning guidance except that the actual memory used can be much larger if multiple operations run in parallel for single queries, if multiple queries run in parallel (which is expected), and so on.14

## Measuring Query Execution Time

**SQL Server**

SQL Server provides at least eight different ways to measure query execution time.[[15]](#footnote-15) Not all are straightforward to access or clearly explained, unfortunately. The simplest and most straightforward metric to access is the “Elapsed time” property of a query in the Properties panel in SQL Server Management Studio. This measure provides millisecond precision.

**Postgres**

Postgres provides much simpler (and more limited) facilities for timing query execution. Postgres itself provides the ability to time queries internally through “EXPLAIN ANALYZE”[[16]](#footnote-16), and the canonical terminal client, psql, also provides the “\timing on” option. The latter appears to be a close analog to SQL Server’s “Elapsed time” property.

**Choosing an execution time metric**

More important than the accuracy or precision of any measurement by either DBMS or client is its comparability to measures provided by the competing system. If I find a straightforward way to retrieve actual plan execution time from SQL Server before I begin benchmarking, I will consider using that measure and Postgres’s “EXPLAIN ANALYZE” measurement. These are likely to be the most accurate, directly comparable measures. Otherwise, I will use SQL Server’s “Elapsed time” and compare it to psql’s “\timing on” option.

Based on the information I have gathered, I feel using ODBC and writing my own scripts to automate testing carries too great a risk of introducing invisible error into the raw data. I consider this to be especially problematic because the SQL Server connector is proprietary and likely to be faster than the Postgres ODBC, which may skew results in favor of SQL Server. In the interest of minimizing error in the raw data, I will perform all testing manually, which will limit the scope of testing I can achieve in the time allotted. Although Microsoft SQL Server Management Studio may employ a higher-performance connector than psql, these two clients are likely to be well-optimized client implementations for their respective systems. I believe this makes them the best clients to use in benchmarking.

# Performance experiments

For all of the experiments below, the following conventions apply:

* In data set specifications, “WB@X rows” means a single table generated via the Wisconsin Benchmark data generation specification from part 1 using a table size of X rows.
* For Postgres, shared\_mem and work\_mem will be adjusted significantly to take better advantage of the test system’s hardware. The exact adjustments are not known at this time, but initial guesses are shared\_mem = 6.4 GB (40% of the test system’s 16 GB of RAM) and work\_mem = 1.6 GB (25% of shared\_mem). To the best of my knowledge, SQL Server will need no adjustments to take full advantage of the test system.
* All data set sizes are estimates. The exact sizes of data sets will be adjusted during part 3 of the project in order to (a) ensure that query execution times are high enough to measure with confidence, (b) keep query execution times low enough to accommodate the high volume of testing, and (c) attempt to ensure that systems perform either in-memory or on-disk operations as needed. (c) is required because Postgres requires manual tuning of its memory model and such tuning is beyond the scope of this project and beyond my knowledge; comparing performance between SQL Server’s in-memory operations and Postgres’s on-disk operations will skew results.
* For each data point, I will run the experiment 5 times, recording each value. I will remove the min and max of the five and average the rest.

## Experiment 1: the 10% rule of thumb

The 10% rule in query planning states that at approximately 10% selectivity, an unclustered index has about the same performance as no index at all, and above this percent, an unclustered index will perform worse. This experiment investigates how the DBMSes handle varying selectivities on a field with an unclustered index.

**Data sets:**

* WB@10 million rows, with an unclustered B-tree index on onePercent

**Queries:**

SELECT \*

FROM **TableName**

WHERE onePercent < **selectivityPercent**;

Since onePercent varies from 0 through 99, **selectivityPercent** values from 0 through 100 will match 0% through 100% of rows, respectively. I will vary **selectivityPercent** across a range that includes 10%. I will choose the exact **selectivityPercent** values during the experiment in order to illuminate any apparent inflection points in the plots, should they arise.

**Expected results:**

I expect both DBMSes to transition from degrading performance on an unclustered index to linear performance using sequential scans at some point. I expect Postgres to make a less optimal choice of cut-off than SQL Server. I expect a jump in the graph where the execution strategy changes.

**Notes:**

This process is extremely labor-intensive. Each **selectivityPercent** value requires 10 raw data points (5 per DBMS). It is highly unlikely that I will be able to check the range of 5% to 40% in increments of 1% as suggested in class, since this would require 360 raw data points.

## Experiment 2: insertions ordered on a clustered index

As discussed in the research above, SQL Server uses certain index types as its physical storage schemes. Specifically, it supports clustered indexes and will use a clustered index as the physical storage scheme for the table. While the benefits of clustered indexes are well-known, maintaining physical ordering during insertions is a tricky problem. This experiment aims to quantify the performance impact of clustered indexes on bulk insertions and to compare clustered index insert performance to Postgres’s unclustered approach.

**Insertion method:**

In terms of insertion method, there are at least three ways to structure such a test:

* Inserting one row at a time
* Inserting many rows in one insert statement
* Bulk loading rows from a file

Inserting one row at a time is useful in a test between different SQL Server table configurations, but when comparing to Postgres, I am concerned that the high volume of insert operations and the small size of each may cause me to inadvertently compare costs like query optimizer run-time and connector efficiency rather than query execution time.

Bulk loading rows from a file is an entirely different process that is optimized separately. This process is not what I wish to test.

Inserting many rows in one insert statement will likely allow SQL Server to optimize the insertion of the rows, for instance, by sorting the rows before inserting them. Although this dilutes the efficacy of this test, I consider it to be the best option, since inserting one row at a time would likely skew the results vs. Postgres. However, it would be interesting to see another experiment that tested SQL Server with clustered index vs. SQL Server with heap file using single-row insert statements, which might be a more realistic scenario.

**Data sets:**

All data sets will be inserted into newly created tables. All data sets will be the same size (initial guess: 10,000 rows), inserted into tables pre-populated with WB@1 million rows, with a B-tree index (clustered in SQL Server, unclustered in Postgres) as specified. The order of insertions will vary:

* Rows sorted by unique2, ascending, with index on unique2
* Rows sorted by unique2, descending, with index on unique2
* Rows sorted by unique2, ascending, with index on unique1 (i.e. pseudorandom order with respect to the index)

**Queries:**

INSERT INTO **TableName** (unique1, unique2, …, string4) VALUES

…

**Expected results:**

I expect Postgres to be significantly faster on insert in all cases. Either the ascending or descending sort with index on unique2 may be exceptionally slow, and the other may be relatively fast, but I anticipate that SQL Server will probably optimize insertions in all cases, e.g. by sorting the incoming data set before inserting it.

## Experiment 3: optimizing 3-way joins

As we discuss in class, estimating the result size of a join operation is a very difficult problem. In this experiment, I will present each system’s optimizer a join optimization problem to see how it responds. I will join three tables with two joins. The tables will all be the same size, but one join will have a small result set and one will have a large result set. I will provide three queries to each DBMS: one that lists the smaller join first, one that lists the larger join first, and one that provides an equivalent query using WHERE predicates instead of the JOIN keyword. This will allow me both to compare the two systems and to see whether each system is able to intelligently optimize these equivalent queries. The latter will be especially helpful if one system turns out to simply be categorically faster than the other throughout my benchmarking.

Note that in every case, each row in one table will be joined with multiple rows in the other tables.

**Data Sets:**

This experiment has one data set comprising three tables:

* A: WB@1,000 rows
* B: WB@ WB@1,000 rows
* C: WB@1,000 rows

Aside from the primary key, no other indexes are present.

**Queries:**

1. SELECT \*  
   FROM A JOIN B ON A.tenPercent = B.tenPercent  
    JOIN C ON B.onePercent = C.onePercent
2. SELECT \*  
   FROM B JOIN C ON B.onePercent = C.onePercent   
    JOIN A ON A.tenPercent = B.tenPercent
3. SELECT \*  
   FROM A, B, C  
   WHERE A.tenPercent = B.tenPercent AND B.onePercent = C.onePercent

**Expected results:**

I expect that both DBMSes’ query optimizers will find relatively similar, perhaps even identical, query plans for all three queries, since they are straightforward and simple. I expect SQL Server to be faster than Postgres.

## Experiment 4: single- and multi-column indexes

SQL Server[[17]](#footnote-17) and Postgres[[18]](#footnote-18) follow distinct but similar rules for using multi-column indexes. In class, we learned that a B-tree index on multiple columns is only useful if the search key columns form a prefix, that is, a leading subset, of the index key columns. Common sense tells me that such an index will be beneficial if it indexes a relatively small subset of the columns on the table and that it will be slower than an index on a single column for a query with a single-column selection predicate. A situation with a single-column index and a multi-column index that shares its first column with single-column index might arise, for instance, when a column with a unique or primary key constraint appears as the first column in a multi-column index that we use to optimize a common query with WHERE predicates on multiple columns.

This experiment asks two questions. First, what is the performance difference between a multi-column index and a single-column index? Second, will both DBMSes intelligently choose between them, if both exist and are suitable for the query in question?

In order to avoid SQL Server’s clustered indexes, which may provide a significant advantage over any unclustered indexes, I will avoid primary key columns.

**Data sets:**

I will test the query with the following configurations:

* Control: WB@5 million rows, no indexes except primary key
* Single-only: WB@5 million rows, single-column unclustered B-tree index on unique1
* Multi-only: WB@5 million rows, multi-column unclustered B-tree index on (unique1, unique2, four, two)
* Both: WB@5 million rows with both indexes above

**Queries:**

SELECT \*

FROM **TableName**

WHERE unique1 = **some random, constant number**

**Expected results:**

I expect **control** to be slowest, **single-only** to be fastest, and **both** to be similar to **single-only** on both systems. I expect **multi-only** to be significantly slower than **single-only** and much faster than **control**.

# Lessons Learned & Issues Encountered

I came into this project with significant experience using Postgres and no experience using SQL Server. I learned a great deal about each system on its own and about the differences between them, and a lot of what I learned surprised me.

When I researched the index types that each DBMS supports, I was astonished to see the outstanding level of professionalism and technical excellence that SQL Server clearly displays. It supports row-wise, column-wise, and in-memory data stores (with an option to store the data on disk or to have purely transient tables). Both column-wise and in-memory data stores can achieve many times the performance of row-wise data stores for broad ranges of common use cases. By contrast, Postgres only supports unclustered, row-wise stores out of the box, though some third-party vendors offer extensions for other use cases. The indexes reflect the products’ different levels of polish as well: while SQL Server goes well above and beyond the typical assortment of indexes with thoughtful, useful, and novel extensions, Postgres simply supports the most common index types plus a few general-purpose or niche index methods.

In terms of index selection in particular, my distinct impression is that SQL Server is designed for production, large-scale databases, while Postgres has received a great deal of attention from contributors in the scientific and academic communities. For instance, whereas SQL Server supports the straightforward and frequently-useful-in-production index with included columns, Postgres supports defining more science-and-research-oriented indexes on custom data types.

Coming into this project, I knew that different database vendors augmented SQL with their own extensions and modifications, and I knew that the vendor-specific clients also supported their own, unique features. Perhaps my steepest learning curves have come from learning the seams between standards and vendor-specific details. I had previously simply treated DBMSes as services that responded to DDL and DML commands. Researching, using, and comparing two DBMSes side-by-side has taught me a lot about the real differences between them.

Overall, I have been thoroughly impressed by SQL Server thus far, which confirms what I have heard. I have found one, notable exception to this, though. To my great surprise, examining query execution time in SQL Server is much harder than in Postgres! The latter’s simplicity steals the show in this case. However, I do not know how accurate either vendor’s measurements may be.

1. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/indexes?view=sql-server-2017 [↑](#footnote-ref-1)
2. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-2017 [↑](#footnote-ref-2)
3. https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/introduction-to-memory-optimized-tables?view=sql-server-2017 [↑](#footnote-ref-3)
4. https://www.postgresql.org/docs/current/sql-cluster.html [↑](#footnote-ref-4)
5. https://www.postgresql.org/docs/current/indexes-types.html [↑](#footnote-ref-5)
6. http://www.sai.msu.su/~megera/postgres/gist/doc/intro.shtml [↑](#footnote-ref-6)
7. https://www.pgcon.org/2011/schedule/attachments/197\_pgcon-2011.pdf [↑](#footnote-ref-7)
8. https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins?view=sql-server-2017 [↑](#footnote-ref-8)
9. https://www.postgresql.org/docs/11/runtime-config-query.html [↑](#footnote-ref-9)
10. https://www.postgresql-archive.org/Partition-wise-join-for-join-between-declaratively-partitioned-tables-td5907846.html [↑](#footnote-ref-10)
11. https://docs.microsoft.com/en-us/sql/relational-databases/memory-management-architecture-guide?view=sql-server-2017 [↑](#footnote-ref-11)
12. https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/server-memory-server-configuration-options?view=sql-server-2017 [↑](#footnote-ref-12)
13. https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/buffer-pool-extension?view=sql-server-2017 [↑](#footnote-ref-13)
14. https://www.postgresql.org/docs/current/runtime-config-resource.html [↑](#footnote-ref-14)
15. https://www.scarydba.com/2018/08/13/measuring-query-execution-time/ [↑](#footnote-ref-15)
16. https://www.postgresql.org/docs/current/sql-explain.html [↑](#footnote-ref-16)
17. https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-2017 (see Column Considerations in the section General Index Design Guidelines) [↑](#footnote-ref-17)
18. https://www.postgresql.org/docs/11/indexes-multicolumn.html [↑](#footnote-ref-18)