

# A Brief Comparison of Two Enterprise-Class RDBMSs

Andrew Figueroa

Department of Computer Science  
Western Connecticut State University  
181 White St. Danbury, CT 06810  
figueroa039@connect.wcsu.edu

Steven Rollo

Department of Computer Science  
Western Connecticut State University  
181 White St. Danbury, CT 06810  
rollo003@connect.wcsu.edu

## ABSTRACT

This paper is a report from a student-developed study to compare Microsoft SQL Server and PostgreSQL, two widely-used enterprise-class relational database management systems (RDBMSs). The study followed an introductory undergraduate course in relational systems and was designed to help us gain practical understanding of specific DBMSs. During this study, with input from a faculty advisor, we implemented three non-trivial schemas in each system, identified 25 common database design, development, and administration activities while implementing the schemas, and compared the support each system offers to carry out the identified activities. Where relevant, we also compared each system against the SQL standard.

In this report, we present a summary of the similarities and differences we found between the two systems and briefly discuss the “technical suitability” of PostgreSQL to enterprise applications. Although this report is not comprehensive and is too general to comment on the suitability of either system to a specific enterprise application, it can nevertheless provide an initial set of considerations and criteria to choose a system for most enterprise applications.

## Categories and Subject Descriptors

H.2.4 [Database Management]: Systems – *Relational databases*.

## Keywords

Microsoft SQL Server 2016 Express, PostgreSQL 9.6, Relational Database Management System (RDBMS), Schema migration.

## 1. INTRODUCTION

For an undergraduate research project, we sought to gain a better practical understanding of relational database management systems (RDBMSs) used to build enterprise applications. Specifically, we studied Microsoft SQL Server 2016 Express Version 13.0.4001 (MSSQL) and Postgres Version 9.6.3 (Postgres), and their respective SQL implementations Transact-SQL (T-SQL) and PL/pgSQL (pgSQL).

Our study proceeded in three broad steps: implementing schemas and queries for three non-trivial applications in each selected DBMS, identifying common database-related activities, and comparing the facilities each DBMS offers to carry out the identified activities.

We began the study by porting to both MSSQL and Postgres the following schemas originally designed for Oracle Database:

This paper is the product of a student-directed study advised by Dr. Sean Murthy at Western Connecticut State University. The goal of the study is to better understand the differences between enterprise-class DBMSs. MSSQL and Postgres are the two DBMSs this study focuses on. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2017 Andrew Figueroa, Steven Rollo, Sean Murthy. CC BY-NC-SA 4.0.0.4001 (MSSQL) and Postgres Version 9.6 (Postgres), and their respective SQL implementations Transact-SQL (T-SQL) and

**Advert:** This schema describes a system for managing advertisement orders for a fictional newspaper.

**Shelter:** This schema describes a system for managing animal care and adoptions at a fictional animal shelter.

**Babysitting:** This schema describes a fictional babysitting co-op where members earn and use credits through babysitting each other’s children.

**Classroom:** This schema represents an environment where students in a course are allowed appropriate access to a DBMS in order to experiment and complete assignments. Instructors for this course also have access to the student’s work inside of the DBMS.

As we ported the schemas and queries, we observed 26 distinct database activities that commonly occurred. We categorized each such activity as a design, development, or administration activity. We considered as *design* any feature that influences how a schema is created at the logical or physical levels (for example, data types and indexes), whereas we classified as *development* any direct management and manipulation of data (“DML queries”). We labeled activities involving the operation and maintenance of the DBMS as *administration*. It is apparent from these definitions that there is some overlap between each category, but we placed in the category it is most relevant to.

With a list of database activities and a classification in hand, we then studied the details of how MSSQL and Postgres support the activities and also compared the approaches the two systems take. Where relevant, we also compared each system against the SQL standard.

The remaining sections of the paper provide the details, but at a high-level, we conclude from this study the following: Both MSSQL and Postgres include competitive features to support common database activities involved in building enterprise systems, with MSSQL taking a more “implementer friendly” approach. Postgres is quite suitable (technically speaking) for enterprise applications, but its lack of certain features may mean some activities require additional implementation effort. Lastly, Postgres’s SQL implementation is very close to the SQL standard, whereas MSSQL is not. However, standards-compliant code is often not portable and at times can be quite inefficient.

The rest of this paper is organized as follows: Sections 2, 3, and 4 introduce the design, development, and administration activities, respectively. Additionally, these sections detail the differences found between the two DBMSs for each activity. Finally, section 5 summarizes the differences found between MSSQL and Postgres, based on the activities we identified. We then summarize how the SQL implementation of each DBMS compares to the SQL standard, and discuss the technical suitability of PostgreSQL to enterprise applications.

The SQL implementations of schemas and queries for both MSSQL and Postgres, as well as the examples in this paper, are all available in a public Git repository [8].

SQL:1999 Standard	Microsoft SQL Server	PostgreSQL
INTEGER SMALLINT BIGINT (all signed)	INTEGER (4B) SMALLINT (2B) BIGINT (8B) TINYINT (unsigned, 1B)	INTEGER (4B) SMALLINT (2B) BIGINT (8B)
NUMERIC DECIMALDECIMAL	NUMERIC, DECIMAL (up to 38 digits of precision)	NUMERIC, DECIMAL (up to 131072 digits before decimal, up to 16383 after)
REAL DOUBLE PRECISION FLOAT	REAL (identical to FLOAT (24)) FLOAT (n) (1-24: 7 digit precision, 25-53: 15 digit precision)	REAL (usually IEEE single) DOUBLE PRECISION (usually IEEE double) FLOAT (n) (1-24: single, 25-53: double)
CHAR VARCHAR CLOB	CHAR VARCHAR VARCHAR (max) (approx. 2GB max)	CHAR VARCHAR TEXT (approx. 1GB max)
DATE (year 0001-9999)	DATE (years 0001-9999, 3B)	DATE (years 4713 B.C.E - 5874897 C.E., 4B)
TIME (6 fs)	TIME (7 fs, 3-5B)	TIME (6 fs, 8 or 12B, time zone)
TIMESTAMP (year 0001-9999; 6 fs)	DATETIME (years 1753-9999, 3 fs, 8B) DATETIME2 (years 0001-9999, 7 fs, 6-8B) SMALLDATETIME (years 1900-2079, 0 fs, 4B) DATETIMEOFFSET (years 0001-9999, 7 fs, 8-10B, time zone)	TIMESTAMP (years 4713 B.C.E - 294276 C.E., 6 fs, 8B, time zone)

**Table 1. Common data types and their implementation details. Numbers with a “B” suffix denote size in bytes; “fs” stands for fractional seconds. SQL:1999 does not define sizes and value ranges. Values of numeric types are always signed.**

## 2. DESIGN ACTIVITIES

For the purposes of this study, we categorized activities under *design* if they would be heavily involved in schema management. In general, these activities do not directly affect the creation of SQL queries, but rather, decisions that would be made during the creation of logical and physical schemas, and soon after they have been created.

### 2.1 Field Design

We studied several aspects of field design, chief among them being data types related to three common kinds of data: numeric, character, and date-time. Table 1 summarizes the relevant data types defined in the SQL standard and their equivalents in MSSQL and Postgres.

**Numeric fields:** The SQL standard describes two categories of numeric data: *exact numerics* and *approximate numerics*. An exact numeric is a number defined with a specific precision. The types INTEGER, SMALLINT, and BIGINT represent integral numbers. The types NUMERIC and DECIMAL represent numbers with fractional parts and permit the designer to choose *precision* (total number of digits) and *scale* (number of fractional digits). Approximate numerics include REAL, DOUBLE PRECISION, and FLOAT [1§2.4]. These approximate numerics are generally floating-point numbers. The precisions of REAL and DOUBLE PRECISION are implementation defined and static, whereas the FLOAT type is designed to allow the user to specify the necessary precision. The standard allows the precision of any of these exact or approximate numeric types to be implementation-defined. The only restriction is that SMALLINT be less than or equal to INTEGER and that BIGINT be equal to or larger than INTEGER. It also states that NUMERIC must always be represented with the exact precision requested, while DECIMAL can be managed with a precision greater than what was specified. Another restriction is that DOUBLE PRECISION should be of a greater precision than that of the REAL

type. However, the standard does not specify how much greater the precision should be.

Both MSSQL and Postgres fully support the SQL standard-for exact numeric types, MSSQL also provides a TINYINT type for integral values ranging from 0 to 255. Both implementations treat the NUMERIC and DECIMAL data types as synonyms, but each implementation affords a different number of maximum digits [2§Decimal and Numeric, 2§int, 3§8.5].

For approximate numerics, MSSQL does not define a DOUBLE PRECISION data type, but does provide REAL and FLOAT types. This adaptation does not change the precision or range available since the FLOAT type provided by MSSQL has a maximum range identical to that of the DOUBLE PRECISION type that Postgres provides (when using the IEEE 754 standard in Postgres) [2§Float and Real]. Postgres follows the SQL standard and provides an implementation of all three approximate numeric types. However, the precision used depends on the environment in which the DBMS is set up. In most cases, REAL and DOUBLE PRECISION follow the IEEE standard 754 Binary Floating-Point Arithmetic, representing the single-precision and double-precision parts of the standard, respectively. This is the expected behavior as long as "the underlying processor, operating system, and compiler support it". [4§8.1].

**Character fields:** Two main character types are defined in the SQL standard: CHAR and VARCHAR, where CHAR values are fixed length (right-padded with spaces), and VARCHAR values have a variable length (up to a user specified maximum length, after which the string is truncated). The standard also defines the CLOB (character large object) type to store large strings of characters without a maximum length. The character sets and maximum size of these types are implementation defined. MSSQL and Postgres both provide CHAR and VARCHAR types, as expected. However, MSSQL does not define the CLOB type, but permits storing large strings via the type VARCHAR (max). Postgres too does not define the CLOB type, but provides the synonymous type TEXT.

**Date and time fields:** The SQL standard defines three basic date and time types: DATE, TIME, and TIMESTAMP [1§2.4]. MSSQL closely follows the SQL recommendation for the DATE and TIME types with years ranging from 0001 to 9999 C.E. for DATE and 7 fractional seconds for TIME. However, MSSQL's uses DATETIME as a counterpart to TIMESTAMP. This is an important distinction because MSSQL does have a data type named TIMESTAMP, but is used for an entirely unrelated purpose [2§timestamp (T-SQL)]. Additionally, DATETIME is not completely equivalent to the standard's TIMESTAMP, as it can only store years from 1753-9999, and only allows up to 3 digits for fractional seconds (compared to 6 in both the SQL standard and Postgres). In addition to these three types, MSSQL also has other types available for storing date and time. These are SMALLDATETIME (years 1900-2079, 1 minute precision, 4 bytes), DATETIME2 (years 0001-9999, 6 fractional seconds, 6-8 bytes) and DATETIMEOFFSET, which is identical in precision and range to DATETIME2, but is the only datetime type that allows a UTC time offset (time zone) [2§Date and Time]. In general, it is recommended to use DATETIME2 over DATETIME whenever possible, as it has an expanded range and precision, and may use less storage space, depending on the precision specified.

In contrast, Postgres fully implements the three types, along with TIME and TIMESTAMP's WITH TIME ZONE derivatives. Postgres also provides TIMESTAMPTZ as an abbreviation for TIMESTAMP WITH TIME ZONE. One deviation, or perhaps, extension, from the standard is the range of years able to be stored in the DATE and TIMESTAMP types. The SQL standard states the year value is to be exactly 4 digits in length, with years ranging from 0001 to 9999. However, Postgres supports years ranging from 4713 B.C.E to 5874897 C.E. (294276 C.E in TIMESTAMP) [4§8.5]. See Figure 1a and 1b for an implementation of a date and time type from the Babysitting schema in both MSSQL and Postgres.

## 2.2 Computed Columns

Computed columns are a way of representing the results of an expression as an attribute of a table, rather than needing to use the expression in every related query. This process is not explicitly covered by the SQL standard. MSSQL offers this functionality, while Postgres does not. The simplest way to obtain similar, although not identical, functionality in Postgres is to use a view. If it is not possible or desirable to use a view, then there is also a method to perform this by using a row level trigger on a column which users do not have INSERT or UPDATE permissions of. A shortened example from the Babysitting schema where we implemented a computed column in both DBMSs (by using a view in Postgres) is provided in Figure 1a and Figure 1b.

In MSSQL, since the column is defined as being the result of an expression, the computed column cannot be modified directly using an INSERT or UPDATE query. Also, certain expressions may limit the ability to use the computed column in a similar manner to a regular (non-computed) column. For example, a non-deterministic expression cannot as a part of a UNIQUE or PRIMARY KEY constraint. In Postgres, any expression that would normally be permitted as a part of a SELECT statement on the underlying table can be used. MSSQL also allows any expression, so long as it is not a subquery. As a result, computed columns are only allowed to reference the columns of the table where it resides, and not those of other tables.

```
CREATE TABLE mssql_test (
    SID INTEGER PRIMARY KEY
    SStart DATETIME2,
    SFinish DATETIME2,
    SLength AS (ROUND(DATEDIFF(mi, SStart,
        SFinish)/ 60.0, 0)) );
```

**Figure 1a. Implementing a computed column in MSSQL**

```
CREATE TABLE pg_test_tab (
    SID INTEGER PRIMARY KEY,
    SStart TIMESTAMP,
    SFinish TIMESTAMP );

CREATE VIEW pg_test_view AS (
    SELECT SID, SStart, SFinish,
        (SFinish - SStart) as SLength
    FROM pg_test_tab );
```

**Figure 1b. Implementing a computed column in Postgres**

When using a view to replace a computed column, no special considerations are necessary and it functions like any other view. However, MSSQL's implementation carries a couple of considerations during use compared to Postgres' implementation. For instance, these computed columns cannot be modified directly through INSERT or UPDATE statements. Also, by default the computed column is a virtual column, and as a result, the values are not stored. This can be changed by marking the column as PERSISTED [2§Specify Computed Columns].

## 2.3 Index Design

Despite not being strictly defined in the SQL standard, indexes comprise a significant portion of schema design. They provide necessary optimizations and are frequently used by internal DBMS operations. However, this lack of a specification by the SQL standard does not necessarily mean that their implementations differ greatly. Rather, both DBMSs implement indexes in a relatively similar manner. Apart from some syntactic differences, the creation and behavior of indexes in both DBMSs is similar, meaning that most indexes can be designed without needing much modification when moving from one DBMS to the other. Both MSSQL and Postgres allow the creation of clustered and non-clustered indexes. In both DBMSs, changing the type of index affects the optimal data structures and algorithms associated with each index. One difference that could be considered in this area is that MSSQL stores most "regular" indexes using a B-Tree, which cannot be manually changed without changing the type of index [2§Types of Indexes]. Postgres instead allows the internal storage method to be manually set for each index. The storage methods for indexes provided by Postgres include B-tree, hash, GiST, SP-GiST, GIN, and BRIN [4§11.2]. A custom index storage method can be defined, but the Postgres documentation warns that this process can be rather difficult. Although "typical" indexes are implemented similarly, some differences arise when needing to create special kinds of indexes. The three special kinds of indexes that we focus on are indexes with included columns, filtered indexes, and indexes on expressions.

**Included Columns:** Indexes with included columns are indexes where in addition to the key attributes (or expressions, as we discuss later), one or more columns are also included and stored with the index. However, these extra columns are not used to organize the index, making them non-key columns in relation to the index. By adding included columns, a well-designed index can eliminate

the need for reading data pages for a significant portion of the workload, greatly improving performance for queries that benefit from this covering index. In MSSQL this is performed by adding the `INCLUDE` keyword after the table that the index is created on, followed by the desired included columns in parenthesis, as in Figure 2a. [2§Index w Included Columns]. Adding a column in this manner is less computationally expensive than adding it as a key column for the index. Since the index is not sorted or managed in relation to these columns, queries that only involve included columns, and not the non-key columns, will not see a performance benefit due to the index. Although Postgres supports indexes with more than one column, it does not support the previously described included columns. Instead, additional columns must be added as key columns for the index (Figure 2b) [4§11.3]. This reduces the net benefit of indexes with multiple columns, as additional computation time is needed to maintain the additional sort order for the index.

```
CREATE NONCLUSTERED INDEX
    IX_dog_arrivalDate
ON dog (arrival_date)
INCLUDE (name, breed)
WHERE arrival_date > '2010-01-01';
```

**Figure 2a. A filtered index with included columns (MSSQL)**

```
CREATE INDEX IX_dog_arrivalDate
ON dog (arrival_date, name, breed)
WHERE arrival_date > '2010-01-01';
```

**Figure 2b. A filtered multi-column index (Postgres)**

**Filtered Indexes:** A filtered index does not include all rows of the table that it is indexing. This reduces the size and the computational overhead of the index, since it does not need to be updated for rows that would not benefit from being indexed. The rows in the index are selected through the value of a specified conditional statement. Both MSSQL [2§Create Filtered Indexes] and Postgres [4§11.8] implement filtered indexes in an identical manner. In both DBMSs, they are created by adding a `WHERE` keyword, followed by a conditional statement. Figure 2a and 2b show a filtered index being created. These filtered indexes are especially useful for tables that have many records, but only a small (but large enough to benefit from indexing) portion are frequently accessed.

**Indexing on Expressions:** The final kind of index that we focus on are indexes on expressions. In addition to indexing on the columns of a table, indexes can also be created on expressions which involve the columns of a table. These are particularly useful for queries that involve sorting or grouping by an expression, especially if the expression in question is computationally expensive and frequently used. MSSQL does not support indexing on expressions in a straightforward manner. Instead, they must be defined indirectly through computed columns (see previous section) in the base table [2§Indexes on Computed Columns]. After creating the computed column on the base table, the index should then be created or altered to accommodate that computed column. Postgres instead allows expressions to be declared directly alongside regular columns when creating or altering the index [4§11.7]. See Figure 3a and 3b for an example of creating an index on an expression.

```
ALTER TABLE dog
ADD weight_celsius AS
    ROUND(weight * 0.45359, 0);

CREATE NONCLUSTERED INDEX
    IX_dog_arrivalDate_DegC
ON dog (arrival_date, weight_celsius);
```

**Figure 3a. An index on an expression in MSSQL**

```
CREATE INDEX IX_dog_arrivalDate_DegC
ON dog (arrival_date,
    ROUND(weight * 0.45359));
```

**Figure 3b. An index on an expression in Postgres**

## 2.4 Querying Metadata

A necessary feature when creating or maintaining database designs is the ability to query metadata. Metadata stored by DBMSs includes many critical details about database objects. For example, the number and type of columns in each table, and the permissions each user has on each table. The ability to perform queries on metadata provides a method to evaluate the current implementation of a database. This metadata can be compared to design documents, and any discrepancies in the design and implementation can be addressed.

MSSQL and Postgres each provide two methods to access metadata. Both DBMSs have at least partial support for the `INFORMATION_SCHEMA`, a set of metadata containing views described in the SQL standard. For ease of writing, we refer to `INFORMATION_SCHEMA` as *info schema*. Additionally, both DBMSs have system catalogs, which provide non-standard method to access metadata. The system catalogs also provide access to metadata that is DBMS specific.

Both DBMSs have at least partial support for the standard *info schema*. In our survey, we focused on views that contain metadata about tables and columns. For example, the `INFORMATION_SCHEMA.COLUMNS` view includes the name, data type, and maximum allowed length of each column in every table (Figure 4). This query works in both T-SQL and *pgSQL*, as they both implement this view.

```
SELECT column_name, data_type,
    character_maximum_length
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'advertiser_t';
```

**Figure 4. A simple `INFORMATION_SCHEMA` query (Advert)**

A more complex query allows us to find the column name, table name, and constraint name of every foreign key constraint in the database (Figure 5). This query also works in both T-SQL and *pgSQL*, and is schema independent.

```
SELECT t.TABLE_NAME, c.COLUMN_NAME,
    c.CONSTRAINT_NAME
FROM
    INFORMATION_SCHEMA.TABLE_CONSTRAINTS t
JOIN
    INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE c
ON t.CONSTRAINT_NAME = c.CONSTRAINT_NAME
WHERE CONSTRAINT_TYPE = 'FOREIGN KEY';
```

**Figure 5. A more complex INFORMATION\_SCHEMA query**

While both MSSQL and Postgres support the info schema, Postgres' support is much more complete. There are 63 views defined in the SQL standard [1§22]. MSSQL only implements 20 of these views, whereas Postgres implements all 63 [2§Information Schema; 4§35]. One specific example is that MSSQL lacks the INFORMATION\_SCHEMA.USAGE\_PRIVILEGES view. Instead, the sys.database\_permissions table contains similar information. The documentation for both DBMSs notes that all DBMS specific metadata is stored in the system catalog. Additionally, Microsoft's documentation warns against using info schema to reliably determine the schema of a table [2§COLUMNS].

MSSQL's lack of support for the info schema is a major detriment to its standards compliance and code portability. While it does implement some of the more commonly used views, the documentation's assertion that these may not be accurate suggests that only the proprietary system catalog should be used. Thus, MSSQL does not support a portable method to query metadata, which is a detriment when considering code portability. Postgres has a major advantage here, since it fully supports the info schema, with the system catalog only providing supplementary metadata.

## 2.5 Views

**Materialized Views:** A materialized view stores the results of a query on the disk as an object in the database. This is in contrast with a non-materialized view, which is completely virtual and not stored. Another major difference is that a materialized view is not necessarily kept up-to-date with the data that it originated from. MSSQL supports materialized views indirectly, while Postgres uses a direct syntax. In order to implement what is similar to a materialized view in MSSQL, a unique clustered index must first be created on a view. These are referred to as Indexed Views [2§Create Indexed Views]. In Postgres, a materialized view is created with a CREATE MATERIALIZED VIEW clause, which is then followed by the appropriate subquery [4§39.3]. Along with being created in a different manner, the two differ in that the MSSQL implementation keeps the information up-to-date with the original data. In Postgres, materialized views need to be updated with REFRESH MATERIALIZED VIEW whenever necessary. If the information should not be kept up to date, then it is always possible to create a new table in the database using the CREATE TABLE AS syntax.

**Updateable Views:** Both T-SQL and pgSQL support updateable views with some limitations. Views that only SELECT data from one source table and have no computed columns are updateable in both DBMSs [2§CREATE VIEW, 3§VI.I]. For example, the view ADVERTISER in the Advert schema is updateable because it selects data from only the ADVERTISER\_T table. The query in Figure 6a will insert a new row into the ADVERTISER\_T table, then update and delete the row through the ADVERTISER view.

Views defined using a JOIN have less ability to be updated. T-SQL is able to perform UPDATE statements on these views, while pgSQL will fail with an error message. Neither implementation permits row deletion on these views.

```
INSERT INTO ADVERTISER_T VALUES (2, 0, 0);

UPDATE ADVERTISER
SET advertiser_id = 3
WHERE advertiser_id = 2;
```

```
DELETE FROM ADVERTISER
WHERE advertiser_id = 3;
```

**Figure 6a. An example of using an updateable view (Advert)**

```
UPDATE INVOICE
SET invoice_date = '2017-02-01';

DELETE FROM INVOICE
WHERE ad_id = 0;
```

**Figure 6b. Example of a query updating the INVOICE view, defined with a JOIN statement (Advert)**

Figure 6b demonstrates performing UPDATE and DELETE statements on the view INVOICE, which is defined using two JOIN statements. The UPDATE in this query will succeed in T-SQL, but fail in pgSQL. The DELETE will fail on both DBMSs.

The documentation for both DBMSs goes into additional detail on the limitations of updateable views. The limitations in T-SQL are based on how the column being updated is derived. Any INSERT, UPDATE, or DELETE statement on a view can only modify columns that are derived from a single base table. Additionally, the column definition cannot include any aggregate functions [2§CREATE VIEW]. For example, T-SQL is unable to update the fully\_paid column in the INVOICE view (Figure 7).

```
UPDATE INVOICE
SET fully_paid = 0;
```

**Figure 7. Example query that updates a column in INVOICE derived from multiple tables (Advert)**

By comparison, pgSQL's updateable views are more limited. The same qualities that make a view not updateable in T-SQL apply to pgSQL. However, pgSQL makes the entire view not updateable, instead of just the offending column [4§VI.I].

## 3. DEVELOPMENT ACTIVITIES

Activities we categorized as *development* concern the logical and physical levels of schema management. These activities frequently involve the use of DML queries, and are heavily dependent on the SQL implementation of the DBMS. T-SQL and pgSQL have many differences that require modified SQL code for the same effect in each implementation.

### 3.1 Common Table Expressions

Both T-SQL and pgSQL support common table expressions using the WITH clause. For non-recursive statements, the two implementations are functionally identical.

However, the syntax for recursive WITH statements differs. T-SQL will evaluate any WITH statement recursively, as long as the code is recursive. pgSQL will only evaluate a WITH statement recursively if the RECURSIVE clause is included. Requiring the RECURSIVE clause is consistent with the SQL standard [1§9.13]. Figures 8a and 8b demonstrate a recursive WITH statement in both implementations.

```
WITH expanded AS (
SELECT NEW.ad_id, NEW.start_date,
CAST(NEW.num_issues AS int) ni
UNION ALL
SELECT ad_id, start_date, ni - 1
FROM expanded
WHERE ni > 1)
```

**Figure 8a. Recursive WITH example (T-SQL)**

```
WITH RECURSIVE expanded AS (
SELECT NEW.ad_id, NEW.start_date,
CAST(NEW.num_issues AS int) ni
UNION ALL
SELECT ad_id, start_date, ni - 1
FROM expanded
WHERE ni > 1)
```

**Figure 8b. Recursive WITH example (pgSQL)**

### 3.2 Query Batch Execution

MSSQL and Postgres interpret queries that have been sent through a client interface differently. MSSQL interprets incoming queries in batches. Microsoft defines a query batch as, “a group of two or more SQL statements or a single SQL statement that has the same effect as a group of two or more SQL statements,” [2§Batches]. Postgres does not have a similar construct, and all queries are essentially evaluated in their own batches.

Thus, when working in an interface such as MSSQL Management Studio, the DBMS interprets any script that results in the execution of two or more SQL statements as a query batch. MSSQL imposes some limitations on what groupings of statements can be in a single batch. The following statements must be executed in their own batch: CREATE DEFAULT, CREATE FUNCTION, CREATE PROCEDURE, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER, CREATE VIEW. Additionally, columns added or modified using an ALTER TABLE statement may not be referenced by other queries in the same batch [2§Batches].

T-SQL does not provide a native statement for explicitly separating a SQL script into multiple batches. To achieve this, Microsoft’s client programs implement the GO command [2§GO]. When the program encounters a GO command, it sends all preceding commands in their own batch, and begins sending all following queries in a new batch. This allows a single SQL script to execute multiple statements that would not be allowed in the same batch.

### 3.3 Text Matching

Despite being a common task, the SQL standard does not define advanced support for text matching beyond two simple wildcard characters and whether matching should be case sensitive. Both T-SQL and pgSQL provide full support for the two wildcards, with the only difference between the two implementations being that T-SQL is case insensitive when using the LIKE keyword, and does not provide a simple method to make the matching case sensitive. pgSQL instead defaults to being case sensitive, which can be disabled through the use of the ILIKE keyword. However, since these facilities are not sufficient for all purposes, both implementations provide extended functionality for text matching through the use of regular expressions. In T-SQL, these expressions still use the LIKE keyword. With these expressions, a single character can be matched to be in or not in a specific range or set of characters [2§Pattern Matching]. pgSQL provides different extensions to text matching.

Additional metacharacters, for example “{n}” which marks the repetition of the previous character n times, and “|”, which provides the ability to match either of two given values, can be used with the SIMILAR TO and NOT SIMILAR TO keywords. Finally, pgSQL also provide full support of POSIX regular expressions with the ~, !~, ~\*, and !~\* operators, depending on whether it should be case sensitive or not, and whether it should be identifying a match or mis-match [3§9.7].

### 3.4 Performance Optimization

Both DBMSs use a cost-based system for determining the best execution plan for a query [2§Hints,3§I.]. The DBMS selects a set of operations that perform the requested query with the lowest calculated cost. The documentation for both states that the DBMS generally selects the best plan, and advise against manually making changes to the plan [2§Hints,3§I.]. Despite this, both DBMSs provide facilities for optimizing query performance, however, their approaches differ significantly.

T-SQL provides *query hints* and *table hints*, which allow the developer to modify the behavior of the query optimizer. Query hints are added in an additional clause to a query, and are only applied to that query. There are several distinct areas of behavior that hints allow you to modify. The JOIN, UNION, and GROUP hints force the optimizer to generate a plan with a specific method of performing JOINS, UNIONS, and GROUP BYs. The MAXDOP hint forces the optimizer to generate a plan with a certain maximum degree of parallelism. Similarly, MAXRECURSION specifies the maximum number of recursive calls the query can perform. The OPTIMIZE FOR hint allows a value for one or more variables to be supplied to the optimizer. It will use these values to generate the query plan, instead of the values that will be used in the query execution. There are also several miscellaneous hints that allow certain query optimizer features to be forced on or disabled.

Query hints also include a subset called table hints, which are applied to a single table in a query. These can force a specific index be used when accessing that table, or modify how the query plan will access the table.

MSSQL provides an additional tool for optimizing performance called *plan guides*. Plan guides force the query optimizer to behave a certain way when it encounters a specific query text [2§Plan Guides]. This is useful in a situation where the exact text of a query cannot be changed, such as when an external application is issuing the query. Plan guides are created using the sp\_create\_plan\_guide stored procedure, which is provided the query text to match and the query hints that should be applied to it. Guides can only be scoped to a single database, so only plans from the current database a query is executing from are eligible to be applied to the query [2§Plan Guides].

MSSQL also provides an additional form of plan guides, *template plan guides*. These allow a plan guide to be applied to any query text that can be parameterized to a certain query, even if the text does not match exactly [2§Create a Plan Guide for Parameterized Queries]. Template plan guides are created by first using sp\_get\_query\_template to get the template for the desired query, and the using sp\_create\_plan\_guide with the parameterized template.

Postgres provides a very different method for performance optimization. There is no facility provided for changing the behavior of the planner on a query by query basis. Instead, Postgres exposes many parameters to change the overall behavior of the planner [4§19.7]. Two types of configuration options are exposed, *method*

configurations and cost constants. Method configurations allow the user to disable certain query execution operations, similar to MSSQL's query hints. Unlike query hints, these options apply to all plans generated by the query planner. The Postgres documentation warns that these options are not intended to be permanent solutions, rather they should only be used for performance testing [4§19.7].

For permanent changes to the planner's behavior, Postgres provides cost constants. These allows the user to set the cost for different internal database operations, which can cause the planner to select different plans.

### 3.5 Dynamic SQL

Both DBMSs support executing a dynamically constructed string as a SQL query. MSSQL provides the EXEC statement and sp\_executesql stored procedure to accomplish this. EXEC will compile and execute a statement string at runtime, while sp\_executesql will take a parameterized query string and the relevant parameters, construct the final query, and then execute it.

```
DECLARE @query_text NVARCHAR(MAX);
SET @query_text = '';

SELECT @query_text += 'SELECT * FROM ' +
table_name + '; '
FROM INFORMATION_SCHEMA.TABLES;

EXEC(@query_text);
```

Figure 9. Example of dynamic SQL in MSSQL (Advert)

Postgres also supports executing dynamic SQL using the EXECUTE statement. Additionally, the format() function can be used to construct a parameterized query, similar to sp\_executesql. EXECUTE can only be used to perform a dynamic query in a PL/pgSQL context. If the query is expected to return a single value, the syntax EXECUTE <query>; INTO <var>; can be used to directly place the result of the query into a variable.

The requirement of executing dynamic SQL in PL/pgSQL contexts can complicate certain queries. For example, the query in Figure 9 can be implemented in PL/pgSQL, as shown in Figure 10. However, the output from the query cannot be easily displayed, because any output from a PL/pgSQL code block must be explicitly returned. Data cannot be arbitrarily output to the console using SELECT statements. This query outputs the text of the generated query for demonstration purposes. Note that although the dynamic query is executed, the output is discarded.

```
DO $$
DECLARE
    query TEXT;
BEGIN
    SELECT string_agg(
        'SELECT * FROM ' || table_name, ';'
    INTO query
    FROM INFORMATION_SCHEMA.TABLES
    WHERE table_schema='public'
    AND NOT table_name ='';
```

```
EXECUTE(query);

RAISE NOTICE 'Query to select from all tables
in database: %I', query;

END$$;
```

Figure 10. Example of dynamic SQL in PL/pgSQL (Advert)

### 3.6 Miscellaneous Activities

We also identified several minor activities related to development. While these do not represent the most important differences, they should be kept in mind during a schema migration process.

**Date Arithmetic:** T-SQL and pgSQL provide different facilities for date arithmetic. T-SQL provides non-standard functions for performing date arithmetic, including DATEADD() and DATEDIFF() [2§Date and Time]. These functions take a date interval, such as day, and two DATETIME values and return the sum or difference of the two dates (Figure 11a). pgSQL follows the standard, and allows for date arithmetic using normal arithmetic operators. Figure 11b performs the same query as Figure 11a in Postgres [4§8.5]. Note that DATEDIFF subtracts the first input value from the second, whereas pgSQL's arithmetic operator subtracts the second operand from the first.

```
SELECT DATEDIFF(day,
    (SELECT MIN(issue_date) first_issue
    FROM AD_RUN
    WHERE ad_id = 0),
    (SELECT MAX(issue_date) last_issue
    FROM AD_RUN
    WHERE ad_id = 0)) + 1 run_length;
```

Figure 11a. Example of DATEDIFF() (Advert)

```
SELECT
    (SELECT MAX(issue_date) first_issue
    FROM AD_RUN
    WHERE ad_id = 0) -
    (SELECT MIN(issue_date) last_issue
    FROM AD_RUN
    WHERE ad_id = 0) + 1 run_length;
```

Figure 11b. Example of pgSQL date arithmetic syntax (Advert)

**Expressions in ORDER BY and GROUP BY:** Both T-SQL and pgSQL allow an identical set of expressions to be a part of an ORDER BY or GROUP BY clause. If an expression uses a function, it must be a scalar function that performs an identical action to every value that is queried. As a part of the Babysitting schema, we were asked to query a list of birthdates and addresses, sorted by day of the year, in order to create a birthday mailing list. In order to perform the sorting, we used an expression in the ORDER BY clause that would sort the list by only the month and day parts of a date. Both the T-SQL and pgSQL implementations used a single function to extract these date parts. However, they used different functions due to differences in the functions available to manipulate dates.

**TRIM Functions:** Postgres provides a TRIM() function. T-SQL does not implement a function called TRIM(). Instead it implements LTRIM() and RTRIM(), which remove whitespace from

the left and right sides of a string, respectively. When used together, they are effectively the same as `TRIM()`.

**T-SQL Stored Procedure Library:** T-SQL provides several built-in stored procedures for retrieving metadata from the system catalog. In particular, we discovered `sp_columns` when looking for a replacement for Oracle's `DESCRIBE` statement. `sp_columns` returns metadata about a table's columns, similar to the info schema query in (Figure 3). Since T-SQL does not fully support info schema, `sp_columns` allows easy access to column metadata stored in the system catalog. Several other stored procedures with similar functionality are provided, such as `sp_tables` [2§sp\_columns].

**SELECT Without FROM:** Another development activity we identified was using a `SELECT` statement to select data not inside the database. This feature is commonly used as a way to output text from a query without necessarily accessing data inside the database. Both T-SQL and pgSQL accomplish this using `SELECT` statements without a `WHERE` clause. This allows a query to output data that is not inside the DBMS.

**Type Casting and Conversion:** Since SQL is a strongly typed language, both T-SQL and pgSQL provide facilities to convert between variable types. T-SQL provides the `CAST` and `CONVERT` statements for explicitly converting between data types. `CAST` provides the SQL standard facility for type conversion, using the syntax `CAST(<var> AS <type>)`. T-SQL will try to convert the supplied value to the requested data type, and throw an exception if the types are incompatible. `CONVERT` behaves similarly, and provides additional facilities for how the cast should be performed. For example, `CONVERT` can cast a `FLOAT` number into a `VARCHAR` with commas and a fixed fractional precision. T-SQL also provides `TRY_CAST` and `TRY_CONVERT`, which return `NULL` if the cast cannot be performed, instead of an exception. Additionally, the MSDN documentation provides a table of which data types can be implicitly converted, explicitly converted, or are incompatible [2§CAST and CONVERT].

Postgres also provides the `CAST` statement for type conversion, with the same syntax. pgSQL will also attempt to implicitly convert types when necessary. It will first find if the operator or function has a version that matches the input type or types. If not, it will attempt to convert one or more of the input values to a compatible type. If one or more of the input values still does not match the required values, an exception is thrown. The same process is followed when attempting to insert values into a table [4§10.1].

**Grouping Sets:** The final minor development activity we found was advanced grouping through the use of `GROUPING SETS`, including the two shorthands for common grouping sets: `CUBE` and `ROLLUP`. Grouping by a grouping set runs the `SELECT` expression on each column/expression in the grouping set, outputting the results for each element of the grouping set. This is an important feature because it allows this grouping to be performed with a single query, rather than relying on several `UNION ALL`s. Despite not being defined in the SQL standard, these are all implemented similarly in both T-SQL [2§Using GROUP BY with ROLLUP...] and pgSQL [4§7.2].

## 4. ADMINISTRATION ACTIVITIES

Activities we categorized as *administration* are those which involve the operations and maintenance of the DBMS, rather than those which deal with data modeling or querying. As we were only dealing with a single user and did not need to perform any maintenance

on the systems, our experience with administration activities is limited to those involved during the setup of both DBMSs on our machines. Other administration activities include those performed via the Data Control Language (DCL) component of SQL, and other management and optimization activities.

### 4.1 Environment Setup

Installation of both DBMSs was rather straightforward. Both DBMSs were installed on machines running Windows. (See section 1.2 for the specific versions of each DBMS used.) MSSQL only provides a single installer [3], whereas Postgres' website provides two installers. One is an "Interactive installer" provided by EnterpriseDB, and the other is a "Graphical installer" provided by BigSQL [6]. We found that the Interactive Installer required additional configuration if command line access using the included `psql` tool was desired. However, no other configuration was needed in either distribution if the graphical `pgAdmin` tool was used. Finally, since Postgres is open source, there is also the option to create an individual or custom build of the software.

While both DBMSs can have all administration activities performed via a command line interface, MSSQL's packaged graphical interface, SQL Server Management Studio (SSMS), is also able to perform all of these activities. However, the graphical interface that is usually packaged with Postgres (`pgAdmin`) cannot perform all available administration activities. This tool is a separate community project, not maintained by the same group that manages the Postgres DBMS. Additionally, it has recently undergone a complete rewrite, as such, some issues are bound to arise. However, these issues are actively being worked on, and the latest two versions (v1.3 and v1.4) contain over 60 combined bug fixes.

### 4.2 Users and Roles

Although the SQL standard gives an outline of how a security and permissions system should be implemented, it leaves the separation between roles (groups) and users (authorization identifiers) up to be implementation defined. MSSQL provides a clear separation between groups and users, while Postgres merges them both into the same model (roles). As such, the concept of roles are different in the two DBMSs. In MSSQL, roles always refer to groups, whereas in Postgres, they may refer to either users or groups. MSSQL also provides integration with Windows accounts, allowing an easier infrastructure setup for organizations who already have a Microsoft Windows domain and organizational system in place. Another difference between the two is that MSSQL requires the creation of a login, which is based at the server level. Following that, logins can then be assigned to users (one login per user per database) [2§Server and Database Roles...]. Postgres does not separate the two, instead, it uses a `LOGIN` attribute to allow a role to connect to the database. The presence or lack thereof can be seen as the equivalent of the separation between users and groups. Other role settings which are based at the server level (such as the maximum number of connections) are also done using attributes when creating or altering a role [4§21.2]. The process of creating users is explored through the Classroom scenario. An example of creating a student and an instructor can be seen in Figure 12a and 12b. In the actual implementation of this scenario, these processes can be performed using stored procedures, rather than the exact statements in these two examples.

### 4.3 Execution Permissions

Although a user with administrative or superuser rights is able to perform most management tasks without needing to switch roles, it is sometimes necessary to do so.



```

CREATE ROLE CS205F17Student;
CREATE ROLE CS205F17Instructor;

CREATE USER [WCSU\Students\Ramsey033];
ALTER ROLE CS205F17Student ADD MEMBER
[WCSU\Students\Ramsey033];

CREATE SCHEMA ramsey033 GRANT SELECT,
INSERT, DELETE, UPDATE;
GRANT SELECT ON SCHEMA ramsey033 TO
CS205F17Instructor;

```

**Figure 12a. Creating a student user based on a windows account (MSSQL)**

```

CREATE ROLE CS205F17Student;
CREATE ROLE CS205F17Instructor;

CREATE USER ramsey033 PASSWORD '50045123';
CREATE USER westP PASSWORD '123999888';

GRANT CS205F17Student TO ramsey003;
GRANT CS205F17Instructor TO westP;

CREATE SCHEMA ramsey003;
GRANT ALL PRIVILEGES ON SCHEMA ramsey003 TO
ramsey003;
GRANT USAGE ON SCHEMA ramsey003 TO
cs205F17Instructor;

```

**Figure 12b. Example of creating a student user (Postgres)**

More often, different permissions or restrictions are needed when executing a procedure, or when a procedure needs to impersonate another role in the database. Another reason to switch execution permissions would be as a part of following the least privilege principle, so users and applications will run with the minimum permissions necessary, and only switch to a role with greater permissions when needed. Both MSSQL and Postgres provide methods of doing so. In MSSQL, the `EXECUTE AS <user>` statement allows this [2§EXECUTE AS], and Postgres permits this with the `SET SESSION AUTHORIZATION <role>` statement [4§SET SESSION AUTHORIZATION]. Both implementations also provide methods of returning to the previous user, with the `REVERT` statement in MSSQL, and `RESET SESSION AUTHORIZATION` in Postgres. With Postgres, it is also possible to return to the default role rather than simply resetting by using `DEFAULT SESSION AUTHORIZATION`. This may or may not be the same role that was being used prior to changing the session authorization. Postgres also allows the ability to only change the current role, and not the current session, through the `SET ROLE <role>` statement, and its correspondent `RESET ROLE`. Different privilege checks are run depending on whether a session or a role is being switched. Also, switching sessions changes the available roles that can be switched to. This Postgres implementation is in line with the statements that the SQL standard recommends [1§14.5]. Both implementations also allow switching sessions (along with roles in Postgres) during a transaction, despite being a restriction suggested by the SQL standard.

## 4.4 Stored Procedures

MSSQL supports separate stored procedures and functions, while Postgres implements stored procedures as a subset of functions [2§CREATE PROCEDURE, 4§41.1]. In MSSQL, functions are intended to perform an atomic set of operations on an input to provide some output, while stored procedures may execute several T-SQL statements in one or more transactions. Stored procedures do support input and output parameters, while functions support input parameters and return values. Stored procedures may call functions, or perform any other T-SQL statements. Functions may not call stored procedures, however, because a function must be an atomic operation, and a stored procedure may perform multiple transactions.

Postgres implements stored procedures as a special case of a function. Stored procedures are functions that have no return statement, and thus can't be used in an expression. Both are created using the `CREATE FUNCTION` statement. While both can contain many pgSQL statements, all functions are treated as atomic, so stored procedures can only use a single transaction.

**Language Support:** MSSQL currently only allows T-SQL statements, or Microsoft .NET Framework common runtime language methods to be run in stored procedures [2§Stored Procedures (DB Engine)], whereas Postgres allows the execution of any language that has been implemented to run in Postgres, including C and R. However, some support for the R programming language is available through an alternate framework [2§SQL Server R Services]. Additionally, it has been recently announced that a newer version of MSSQL (SQL Server 2017) adds support for the Python programming language [7].

## 4.5 Error Handling

The final administrative activity we identified was dealing with exceptional situations during the execution of stored procedures. MSSQL's implementation follows the `TRY...CATCH` and `THROW` convention familiar to most computer programmers. If any execution error with a severity that is greater than 10 occurs during the processing of statements in the try block, control is passed to the catch block. [2§TRY...CATCH]. Although Postgres does not implement these `TRY...CATCH` blocks, the `EXCEPTION` keyword provides similar functionality. The `EXCEPTION` keyword can be used once in each `BEGIN` block (typically towards the end of the block). If an error occurs during the execution of a stored procedure, execution is moved to the statements following the `EXCEPTION` keyword. Here, conditions can be checked, followed by statements that can appropriately handle the exceptional circumstances [4§33.8].

## 4.6 Monitoring User Activity

Another important DBMS activity is the ability to monitor user activity. Many applications have strict requirements on logging how users interact with the DBMS. This monitoring can take place at different levels, for example, logging connections to the DBMS or logging insert statements made on a specific table.

Both MSSQL and Postgres provide triggers to monitor user activity. Triggers are facility that allows a procedure to be executed when some event happens. For example, a trigger could be defined such that whenever an insert is attempted on table x, procedure y is executed.

MSSQL supports three different types of triggers, while Postgres supports only two. Both DBMSs support triggers which execute statements on DML or DDL events. MSSQL supports an additional

type of trigger which executes when a user establishes a connection with the DBMS, called a logon trigger.

**DML Triggers:** Both DBMSs support the execution of triggers when DML statements are evaluated. The MSSQL documentation refers to these as DML triggers, while the Postgres documentation refers to them as regular triggers. Both MSSQL and Postgres require a DML trigger be scoped to a single table or view. This also implies that DML triggers only exist in the scope of a single database.

The trigger implementations in both DBMSs only support a subset of the standard. MSSQL does not support row-level triggers, which deviates from the standard [1§11.3; 2§CREATE TRIGGER]. Neither MSSQL nor Postgres support the REFERENCING clause for specifying an alias for new or old rows [1§11.3]. MSSQL provides two tables, `inserted` and `deleted`, which contain any rows that were inserted or deleted by the triggering statement [2§CREATE TRIGGER]. This also allows statement level triggers to act on a row level basis. Postgres provides the aliases `NEW` and `OLD`, which refer to the new and changed rows, respectively [3§I.]. Postgres also requires that the trigger only execute a single, user-defined function, which deviates from the standard [4§I.]. In practice, however, this is easily overcome by having the function execute any needed statements. Figures 13a and 13b present a DML trigger in each DBMS. Note that the MSSQL trigger is able to operate on all deleted rows in a statement-level trigger using the `deleted` table, and Figure 13b contains both the trigger and function definitions.

```
CREATE OR ALTER TRIGGER
delete_ad_runs_trigger
ON ad
INSTEAD OF DELETE
AS
BEGIN
    DELETE FROM ad_run
    WHERE ad_id = ANY(SELECT ad_id FROM
deleted);
    DELETE FROM payment
    WHERE ad_id = ANY(SELECT ad_id FROM
deleted);
    DELETE FROM invoice_t
    WHERE ad_id = ANY(SELECT ad_id FROM
deleted);
    DELETE FROM ad
    WHERE ad_id = ANY(SELECT ad_id FROM
deleted);
END
```

**Figure 13a. DML trigger example in MSSQL (Advert)**

```
CREATE OR REPLACE FUNCTION delete_ad_runs()
RETURNS trigger AS
$BODY$
BEGIN
    DELETE FROM ad_run WHERE ad_id = OLD.ad_id;
    DELETE FROM payment WHERE ad_id = OLD.ad_id;
    DELETE FROM invoice_t WHERE ad_id =OLD.ad_id;
    RETURN OLD;
END;
$BODY$
LANGUAGE plpgsql;
```

```
CREATE TRIGGER delete_ad_runs_trigger
BEFORE DELETE
ON ad_t
FOR EACH ROW
EXECUTE PROCEDURE delete_ad_runs();
```

**Figure 13b. DML trigger example in Postgres (Advert)**

**DDL Triggers:** Both DBMSs also support triggers that execute when DDL statements are performed. MSSQL refers to these as DDL triggers, while Postgres refers to them as event triggers. MSSQL can execute DDL triggers in the scope of a single database, or the entire server. Event triggers can only be executed in the scope of a database in Postgres.

Since these triggers are commonly used for logging, it is important that they can be executed with the correct permissions. MSSQL allows an `EXECUTE AS` clause in a DDL trigger definition. This causes the trigger procedure to be executed with different permissions than the user causing the trigger to execute. There are three permission types `EXECUTE AS` can use: `CALLER`, `SELF`, or `<user name>` [2§EXECUTE AS]. `CALLER` executes the trigger with the same permissions as the executing user. `SELF` executes the trigger with the same permissions as the trigger owner. Additionally, a login name of the user the trigger will execute as can be provided.

Postgres provides similar functionality with the `SECURITY` clause [4§I.]. Since Postgres triggers always execute a user-defined function, the execution permissions are set on the function itself, not the trigger. There are two forms of the `SECURITY` clause, `SECURITY DEFINER` and `SECURITY CALLER`. `SECURITY DEFINER` executes the function with the permissions of the defining user, while `SECURITY CALLER` executes with the permissions of the executor.

**Logon Triggers:** MSSQL also supports logon triggers. These are similar to DDL triggers, but are only executed when a user logs on to the database, and can only be executed in the scope of the whole server. Logon triggers support the same permissions mentioned previously. Postgres does not provide any functionality similar to a logon trigger, however Postgres does have the ability to log connections in the external sever logs [4§19.8]. While this is much less flexible as MSSQL’s logon trigger, it still provides a method to log connections.

## 5. DISCUSSION

In this final section, we discuss the overall findings of this study. We present a brief summary of our findings, a discussion of the standards compliance of both DBMSs, and detail what conclusions we draw from this study.

**Summary:** Based on our testing of 26 common database activities, we conclude from this study the following: Both MSSQL and Postgres include competitive features to support common database activities involved in building enterprise systems, with MSSQL taking a more “implementer friendly” approach. Postgres is quite suitable (technically speaking) for enterprise applications, but its lack of certain features may mean some activities require additional implementation effort. Lastly, Postgres’s SQL implementation is very close to the SQL standard, whereas MSSQL is not. However, standards-compliant code is often not portable and at times can be quite inefficient.

**Standards Compliance:** MSSQL is clearly less standards compliant than Postgres. MSSQL showed major deviations, such as its lack of support for the complete `INFORMATION_SCHEMA`, and minor deviations, such as allowing recursive CTEs without explicit

declaration. While Postgres is largely standards compliant, there are several areas where both DBMSs deviate. For example, both DBMSs deviate in their implementation of the CLOB data type. MSSQL uses a different syntax for performing the same functionality, while Postgres uses a different name.

Despite being largely standards-compliant, we still identified many differences between both DBMSs. The SQL standard primarily dictates features a DBMS must support. It often does not place strict limitations on the behavior of such features. One example is the difference in fractional exact numerical data type ranges, as the standard does not define what they must be. For example, Postgres supported a greatly expanded range for date data types. Another example is that the SQL standard requires support for views, but it does not define any requirements for the behavior of updatable views [1§4.3]. This allows for the differences we observed between the MSSQL and Postgres implementations of updatable views.

This also allows both DBMSs to implement extended features that are not defined in the standard. Postgres' extensible native language support is an example of one such feature. Many of MSSQL's deviations from the standard are used to provide an extended feature set. For example, included columns and query hints are not defined in the standard, however these are features that are clearly useful for many applications. The clear disadvantage to these features is that they are not defined in the standard. Therefore, an application depending on them becomes locked to platforms that support the required non-standard features.

**Conclusions:** One conclusion that can be drawn from these differences is that standards compliant SQL code is not necessarily portable or optimal. While many of the differences shown may individually be minor, their sum can make schema migration difficult. For example, a schema that heavily relies on computed columns or updatable views may be difficult to migrate to Postgres, despite being otherwise standards-compliant. Likewise, a schema that relies on INFORMATION\_SCHEMA may be difficult to migrate to MSSQL, and may require the use of non-standard features.

To help gauge the overall level of support both DBMSs showed for the activities we studied, we assigned a score for each activity based on the level of support the DBMS had for that activity. A score of two indicates complete, and, if possible, standards compliant support for that activity. A score of one indicates partial or non-standard support for that activity. A score of zero indicates no support for that activity. The score for each activity, as well as the total score for each DBMS can be seen in table 2.

The highest possible score was 52 points, a score of two in each of the 26 activities. MSSQL score slightly higher than Postgres, at 42 points compared to 41. This indicates that overall, both DBMSs had a high level of support for the activities we studied. When MSSQL did not have full support for an activity, it tended to have partial or non-standard support. This is shown by its four scores of one compared to only two of zero. Conversely, Postgres tended to have no support instead of partial support, as evidenced by five scores of zero and only a single score of one. This points to one of the fundamental differences between the two DBMSs. MSSQL is more likely to have support for a given activity, however this comes at the cost of standards compliance. Postgres has support for fewer activities, however those it does support are very likely to be standards compliant.

Activity	MSSQL	Postgres
<b>Design</b>		
Standard Field Types	2	2
Querying Metadata	1	2
Native Computed Columns	2	0
Updatable Views	2	2
Materialized Views	1	2
DML Triggers	2	2
Included Columns in Indexes	2	0
Filtered Indexes	2	2
Expressions in Indexes	1	2
<b>Development</b>		
Common Table Expressions	2	2
Unrestricted Query Batches	0	2
Extended Text Matching	1	2
Dynamic SQL	2	2
Query Hints	2	0
Grouping Sets	2	2
Expressions in ORDER BY and GROUP BY	2	2
Standard Type Casting	2	2
<b>Administration</b>		
DDL Triggers	2	2
Server Level Triggers	2	0
Role-Based Access Control	2	2
Execution Permissions	2	2
Full Linux Support	0	2
Stored Procedures	2	1
User Defined Exceptions	2	2
Plan Guides	2	0
Native Language Extensibility	0	2
<b>Total</b>	<b>42</b>	<b>41</b>

**Table 2. Summary of support for each identified activity**

Using this evaluation, we can determine how suitable Postgres is for enterprise applications. The activities Postgres does support function very well, however, many enterprise applications require features that Postgres does not support, such as logon triggers or included columns in indexes. Additionally, there are few places where Postgres is technically more advanced than MSSQL. Some of these limitations can be overcome with additional development work, such as approximating a logon trigger using Postgres' connection logging feature. We must also consider that because Postgres is FOSS software, there are potential cost savings if it is used over commercial software.

Considering all these points, we can say that Postgres is suitable for enterprise applications from a technical standpoint, although the exact suitability to a specific application will depend on the required features. Also, even if Postgres is missing necessary features, it may still be viable since its FOSS nature makes it possible to extend. However, there may be hidden costs to using Postgres if a large amount of development work is needed to overcome its technical limitations.

## ACKNOWLEDGEMENTS

The authors would like to acknowledge Dr. Sean Murthy, who played a critical role in advising which activities should be covered

in this study. He also provided much assistance with reviewing and improving this paper.

## REFERENCES

- [1] Jim Melton and Alan R. Simon. 2002. *SQL:1999: Understanding Relational Language Components*. Academic Press, San Diego, CA, 32-42.
- [2] Microsoft Corporation. 2017. SQL Server Technical Documentation. MSDN Library. [https://msdn.microsoft.com/library/ms130214\(v=sql.130\).aspx](https://msdn.microsoft.com/library/ms130214(v=sql.130).aspx)
- [3] Microsoft Corporation. 2017. SQL Server 2016 SP1 Express Edition. SQL Server. <https://www.microsoft.com/en-us/sql-server/sql-server-editions-express/>
- [4] PostgreSQL Global Development Group. 2017. PostgreSQL 9.6.2 Documentation. <https://www.postgresql.org/files/documentation/pdf/9.6/postgresql-9.6-A4.pdf>
- [5] PostgreSQL Global Development Group. 2017. License. <https://www.postgresql.org/about/licence>
- [6] PostgreSQL Global Development Group. 2017. PostgreSQL: Windows Installers <https://www.postgresql.org/download/windows/>
- [7] Sumit Kumar and Nagesh Pabbisetty. Python in SQL Server 2017: enhanced in-database machine learning. <https://blogs.technet.microsoft.com/dataplatforminsider/2017/04/19/python-in-sql-server-2017-enhanced-in-database-machine-learning/>
- [8] WCSU-murthys / 2017-Spring-CS299-03. 2017. <https://github.com/WCSU-murthys/2017-Spring-CS299-03>

## APPENDIX

### A. SQL CODE FOR SCHEMA

These schemas are available digitally (as .sql files) on GitHub. Additional working examples for each schema are located in their respective folders. Below is a summary of the schemas and examples in the repository.

GitHub Repository:

<https://github.com/WCSU-murthys/2017-Spring-CS299-03>

#### A.1 Advert

##### A.1.1 T-SQL

Schema File: src/mssql/advert/3\_createAdvertTSQL.sql

Additional working examples for this schema:

1\_ddl-login-triggers.sql

4\_dml-trigger.sql

5\_opsTSQL.sql

6\_dynamic-sql.sql

7\_perfromance-optimization.sql

##### A.1.2 pgSQL

Schema File: src/postgres/advert/3\_createAdvertpgSQL.sql

Additional working examples for this schema:

1\_ddl-login-triggers.sql

4\_dml-trigger.sql

5\_opsTSQL.sql

6\_dynamic-sql.sql

7\_perfromance-optimization.sql

#### A.2 Shelter

The implementation of the Shelter schema is identical between T-SQL and pgSQL

##### A.2.1 T-SQL and pgSQL

Directory: src/mssql/shelter/shelter\_TSQL.sql or src/postgres/shelter/shelter\_pgSQL.sql

#### A.3 Babysitting

##### A.3.1 T-SQL

Directory: src/mssql/babysitting/babysitting\_TSQL.sql

##### A.3.2 PgSQL

Directory: src/postgres/babysitting/babysitting\_pgSQL.sql

#### A.4 Classroom

No schemas available as Classroom is only a hypothetical scenario, not a full implementation.

##### A.3.1 T-SQL

Directory: src/mssql/classroom

##### A.3.2 PgSQL

Directory: src/postgres/classroom