2009

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

A QUANTITATIVE ANALYSIS OF SQL SERVER 2008 CONSTRUCTS

Justin Denning

A Capstone Project Submitted to the
University North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2009

Approved by

Advisory Committee

______________________________	______________________________
Dr. Curry Guinn	Dr. Ling He

____
Dr. Douglas Kline, Chair

A Quantitative Analysis of SQL Server 2008 Constructs. Denning, Justin, 2009.
Capstone Paper, University of North Carolina Wilmington.

Given the large investment companies place on database servers and database management systems, it is vital to keep the database infrastructure operating at peak efficiency. One attractive option for improving efficiency is SQL query tuning as it requires no hardware investment and does not adversely impact the system as other non-hardware tuning such as adding additional indexes. Using an adaptation of the TPC-C benchmarking framework, three phrasings and their alternatives will be tested with control dependent variables of size, indexing, and distribution. The queries to be tested are Join, Subquery, Date Between, Date Part, Merge, and a custom “Upsert” query. The results of the trials will be statistically analyzed in order to provide advice to practitioners on what query phrasing is most appropriate given a specific data scenario.

Table of Contents

Page

PROPOSAL

Section 1: Motivation 1
1.1 Alternatives to Query Modification 3

Section 2: Introduction....................................4

Section 3:Background 5
3.1 Physical Storage....................................5
3.2 Balanced Trees 6
3.3 Fragmentation 8
3.4 Locating Data: Table Scans and Indexes....................................8
3.5 Data Distribution....................................11
3.6 Query Optimizer 13

Section 4: Expert Guidance....................................14
4.1 Vendor Dependent Shops 14
4.2 Custom Deployment....................................15

Section 5: Experiments 16
5.1 Assumptions....................................16
5.2 Setup of Environment....................................16
5.2.1 Server....................................16
5.2.2 Data Model 17
5.2.3 TPC-C Benchmark 18
5.2.4 Data Implementation....................................18
5.2.5 Implementation....................................20
5.2.6 Metrics....................................21
5.3 Join vs. Subquery....................................21
5.4 Date Comparison 23
5.5 Merge 23

EXECUTION

Section 6: Experiment Revisions....................................23
6.1 Size as a variable 24
6.2 TPC Setup....................................25
6.3 Merge 26

Section 7: Experiment Execution 26
7.1 TPC Database....................................26
7.2 Test Execution....................................27

Section 8: Data Analysis 27
8.1 Data Collection and Analysis 28
8.2 Join Versus Subquery 29
8.3 Datepart Versus Date Between 31
8.4 Merge 33

Section 9: SQL Server 2008 Review 35

Section 10: Future Research...36

References...38

Appendixes
A. Tested Queries 39
B. Join Analysis Results...41
C. Date Analysis Results...43
D. Merge Analysis Results 45
E. DATE BETWEEN Execution Plan.......................................47
F. Create TPC Database...48
G. Create TPC Objects..50
H. Run Experiment Job Window...82

Figures
1 Representation of a Data Page...6
2 Balanced Tree 7
3 Query on a Clustered Table..10
4 Non-clustered Index on a Heap 10
5 Clustered Table with a Non-Clustered Index 11
6 Execution Plan..13
7 Test Server Description 16
8 TPC Data Model 18
9 Default Index Set 19
10 Extensive Index Set 19
11 Experiment Matrix...21
12 Figure 8 Revision..22
13 TPC Base DB and 2 Copies for Experimentation 23
14 Results Prepared for Analysis...25
15 Subquery v. Join Average Runtimes.................................27
16 Date Part v. Date Between Average Runtimes 28
17 Merge v. Upsert Average Runtimes...................................29
16 Date Part v. Date Between Average Runtimes 31
17 Merge v. Upsert Average Runtimes...................................34

A Quantitative Analysis of SQL Server 2008 Constructs

Justin Denning

Where performance is measured, performance improves. Where performance is measured and reported, the rate of improvement accelerates.

Thomas S. Monson

ABSTRACT

Given the large investment companies place on database servers and database management systems, it is vital to keep the database infrastructure operating at peak efficiency. One attractive option for improving efficiency is SQL query tuning as it requires no hardware investment and does not adversely impact the system as other non-hardware tuning such as adding additional indexes. Using an adaptation of the TPC-C benchmarking framework, three phrasings and their alternatives will be tested with dependent variables of size, indexing, and distribution. The queries to be tested are Join, Subquery, Date Between, Date Part, Merge, and a custom "Upsert" query. The results of the trials will be statistically analyzed in order to provide advice to practitioners on what query phrasing is most appropriate given a specific data scenario.

SECTION 1: MOTIVATION

The opening bell at the stock market sets in motion 5000 transactions per second funneling through Microsoft SQL Server within NASDAQ's Market Data Dissemination System (MDDS). Every NASDAQ trade is handled by MDDS and the performance of SQL Server is paramount to its efficient workflow. Despite the magnitude of a database management system's impact on application operation, 80-90% of tuning occurs at the application level rather than the database level[4]. Furthermore, 80% of SQL Server performance gains on SQL Server come from making improvements in SQL code, not from devising crafty configuration adjustments or tweaking the operating system. With such great performance gains to be reaped, writing efficient SQL code is of great importance and deserves its due diligence.

E-commerce and global business have made on-demand information and 24/7 data availability a necessity. This places a priority on effective online transaction processing in IT departments across the world, and one area that seems to be overlooked is SQL query phrasing[7]. An optimized query can be orders of magnitudes faster than its untuned counterpart conserving resources and provide an improved user experience. Phrasing optimization requires a balance of complex database factors. The proportion of operations (creates, reads, updates, or deletes) executed should be taken into account. A system that executes reads 80% of the time should be approached differently than a system doing 80% creates and updates. Moving closer to the data, one should examine the metadata of the table's contents to gain insight into selectivity and how to take advantage of indexes. Indexes can have a large influence on query execution time. The query below originated from a order processing system in an online transaction processing (OLTP) environment:

```
SELECT order_item.status FROM order_item
WHERE order_item.status RLIKE ‘[PSO]’
AND order_item.psn != “ANDSUBSTRING(order_item.psn,3,2) != ‘11’
AND order_item.modified_date LIKE ‘2008-10-28%’
```

After examining the environment, the query was replaced with a new query

```
(SELECT status FROM order_item WHERE order_item.status
 LIKE ‘P%‘ AND order_item.psn != “ AND SUBSTRING(order_item.psn, 3, 2) != ‘11’
 AND order_item.modified_date LIKE ‘2008-10-28%’) UNION ALL
(SELECT status FROM order_item WHERE order_item.status
 LIKE ‘S%‘ AND order_item.psn != “ AND SUBSTRING(order_item.psn, 3, 2) != ‘11’
 AND order_item.modified_date LIKE ‘2008-10-28%’) UNION ALL
(SELECT status FROM order_item WHERE order_item.status
 LIKE ‘O%‘ AND order_item.psn != “ AND SUBSTRING(order_item.psn, 3, 2) != ‘11’
 AND order_item.modified_date LIKE ‘2008-10-28%’)
```

While considerably longer, the second query takes advantage of an index on order_item.status and is approximately 20% faster on a relatively small data set.

1.1 Alternatives

An organization should evaluate all options when seeking database performance gains. The first option that many companies choose prematurely is to upgrade hardware. While it will yield immediate improvement, the costs can be staggering. An enterprise level database solution can range between $65,000 and $17 million dollars[9] depending on vendors and number of users supported. Rob Ballantine, A UNC Wilmington database administrator, noted that “bad queries cost money”. Query phrasing can free up resources to stave off hardware improvements yielding a higher return on investment.

Adding indexes can also have a profound impact on a system’s performance. Unfortunately, there may be unintended consequences because adding indexes to accelerate one query may actually slow down many other queries. The more complex the database, the harder it will be to predict the implications of new indexes. Rephrasing individual queries has no effect on the performance of the rest of the system and can achieve similar functionality.

This paper will examine the factors that determine the performance of T-SQL queries, and investigate how two differently phrased queries producing equivalent datasets perform in several scenarios. Our view is that query wording can have a noticeable impact on execution time. In developing a systemized analysis of these queries, we hope to provide guidance to practitioners about when particular phrase compositions have distinct advantages over competing constructs.

SECTION 2: INTRODUCTION

This paper will examine the factors that determine the performance of T-SQL queries, and an experiment will show how two differently phrased queries producing equivalent datasets perform in several scenarios. The phrasings to test are: Join vs. Subquery, Date Part vs. Date Between, and Merge vs. Upsert. To gain a practitioner's viewpoint, three database administrators from varied industries were interviewed, and the visits are described in Section 4. Many factors affect a SQL statement's execution time, and SQL wording cannot be considered in isolation. Section 3 provides background into what factors contribute to SQL phrasing performance, and the variables that should be considered to explain SQL phrasing execution time. The variables that were taken into account are database size, data distribution, indexing, and SQL phrasing. The database structure of the TPC-C benchmark was utilized in order to perform 12 phrasing experiments for every permutation of database size (small, medium, and large), indexing (minimal or extensive), and data distribution (uniform or nonuniform). Each phrasing in Appendix A was tested against its alternative in each of the 12 data scenarios, and our hypotheses were the following:

> **Hypothesis 1**: The Join statement will be faster than the Subquery statement in all tested scenarios.
>
> **Hypothesis 2**: The Date Between statement will be faster than the Date Part statement in all tested scenarios.
>
> **Hypothesis 3**: The Merge Statement will be faster than the Upsert query in all tested scenarios.

Section 5 details the experiments as they were designed in Fall of 2008, and Section 6 chronicles the revisions made for execution in the Spring of 2009. Execution is laid out in Section 7, and the results analyzed and recommendations for practitioners given in Section 8. Finally, Section 9 describes future research on SQL phrasing.

SECTION 3: BACKGROUND

Many factors affect the overall runtime of SQL statements. Chief among them are how the data is physically stored, indexing, data statistics, and SQL Server's Query Optimizer. To construct unbiased experimental scenarios and properly hypothesize the results, a core understanding of these concepts must be established.

3.1 Physical Storage

SQL Server hands out space in blocks known as extents. An extent is the basic unit of storage used to allocate space for tables and indexes and consists of eight contiguous 8K data pages (64K total) as in Figure 1. Once an extent is full, the next record to be inserted will increase the size not just the size of the record but the size of the entire new extent. This may seem like wasteful overhead, but pre-allocating space often saves the time of allocating space for each new record.

Extents are separated into 8 data pages, and rows reside on the pages. The number of pages per extent is fixed, but the number of rows per page depends entirely on the size of the row which may vary depending on its contents. A row sits on exactly one page and may not be divided between pages. In the instances where a record is larger than

8K, the data may span multiple pages and a pointer to the content of a column is stored. There are a number of different page types; we are primarily concerned in this project with data and index pages. Data pages contain the actual data in a table. Index pages hold the non-leaf and leaf level pages of a non-clustered index as well as the non-leaf level pages of a clustered index.

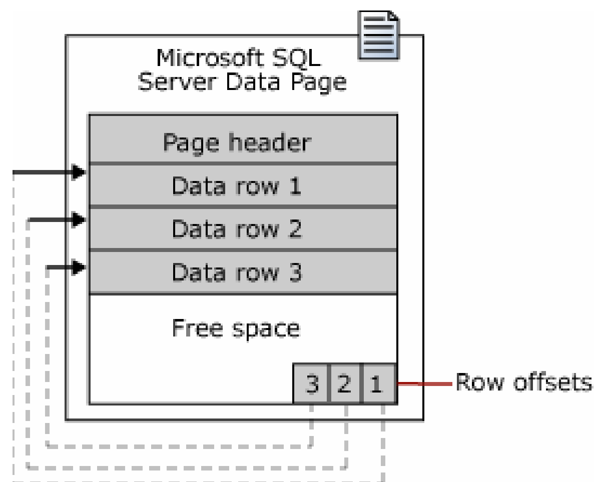


Figure 1: Representation of a Data Page[10]

3.2 Balanced Trees

SQL Server enlists balanced trees (B-Trees) as in Figure 2 to locate and place records. B-Trees have the advantage of being relatively efficient and are generally self-balancing. Balanced means that every time the tree branches, approximately half the data is on one side and half on the other. A search for a record begins at the root node (in SQL Server, a node is an index page), and if the record set is small, may point directly at the actual location of the data[8]. In most cases, we search through the root until we find the last page that starts with a value less than what we are looking for. We

then obtain a pointer to that node and go to that node. In most cases, there is too much data to reference from the root node, so the root will point to intermediary nodes called non-leaf level nodes. These nodes can point to other non-leaf level nodes, or to leaf level nodes- the nodes that contain the reference to the actual physical data. Figure 2 illustrates the idea of root, non-leaf, and leaf level nodes.

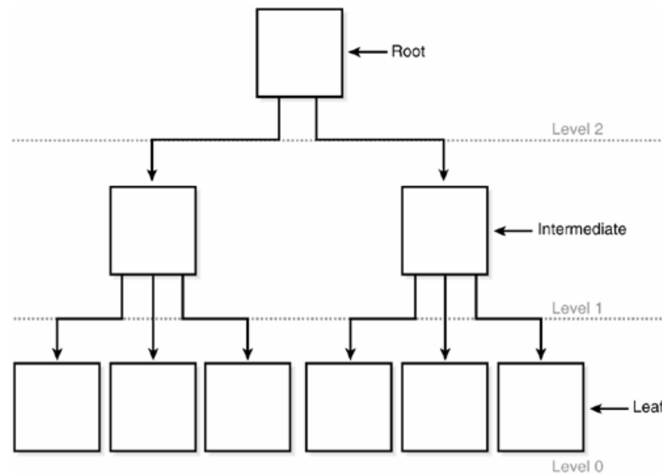


Figure 2: Balanced Tree[10]

Reading from a B-Tree is straightforward, but adding data introduces the idea of balancing. As data is added to the tree, pages will eventually become full, and will need to split. When a page splits, the first half of the data remains on the original page, and the rest of the data goes on a new page yielding roughly a 50-50 distribution. Page splits result in considerable overhead from creating a new page, transferring data from the current page to the new one, adding the new row to a page, and adding another entry in the parent node. The addition of the new entry into the parent node could produce another page split if there is no more space in the parent page. This can cascade all the way up to the root node, causing the root node to become an

intermediary node, and an entirely new root node will be created. These cascading splits will be relatively infrequent, but take significant time cycles to complete.

3.3 Fragmentation

The B-Tree is well suited for adding data, but what happens when the database grows and records are removed? Fragmentation is the unused space that is allocated. If there is only one record on an extent, the extent is still allocated. SQL Server can reuse the space as new data is added. A heavily fragmented database is often an advantage for *INSERT* statement performance. The possibility for page splits is greatly reduced as rows can be inserted in the available unused space rather than adding new pages. However, it is not advantageous for *SELECT* statement performance because of the additional overhead for data retrieval. A heavily fragmented database will have more non-leaf level nodes to traverse than its less fragmented counterpart, resulting in longer seek times. Fragmentation in our experiments should be controlled for an accurate assessment of performance, and SQL Server does provide mechanisms for monitoring and controlling fragmentation. The *fillfactor* option can be set for indexes to set the level of fragmentation, and DBCC SHOWCONTIG will list information for tables and indexes to verify that the set fragmentation level is in place. In this work, all scenarios will have a fixed *fillfactor* of 80%.

3.4 Locating Data: Table Scans and Indexes

Our experiments will focus on SELECT statements and we must pay particular attention to how SQL Server retrieves the data. It may opt to perform table scans or employ

indexes. In a table scan, SQL Server starts at the physical beginning of the table and examines every row looking for matches to the search criteria. This works well for small tables, but quickly becomes time consuming for large tables. To efficiently locate data, we should look to an index similar to the index in the back of a book. Indexes are divided into 2 types, clustered and non-clustered. SQL Server negotiates the B-Tree in different ways to arrive at the end data depending on the index type.

A clustered index is unique for a table, and a clustered table is a table which has a clustered index. There can be only one per table because the data is sorted physically in the same order as the index. In traversing the B-Tree, once the leaf level has been reached, you have the actual data and there is no reference to another location on disk. Figure 3 shows a search for the last name “Rudd” on a clustered table. This can greatly enhance performance by avoiding the overhead of obtaining a reference and locating it. It is particularly advantageous for ranged lookups (finding all the customers in a particular zip code for example) on the clustered index, as the desired data is in an adjoining block. It is important to note that despite their advantages, clustered indexes may not be ideal for every table, particularly where inserts are placed in the middle of the data. This can result in an abundance of page splits in an OLTP environment. Clustered indexes are ideal on columns whose data is ever increasing (such as DATETIME or an identity column) so that new rows are inserted at the end of the table making page splits irrelevant. A clustered table is any table that has a clustered index on it and, conversely a heap table is a table that does not have a clustered index. In heap tables, a row ID is created to represent an extent, page, and row that corresponds to the physical location of the data.

```
SELECT lastname, firstname FROM member WHERE lastname = 'Rudd'
```

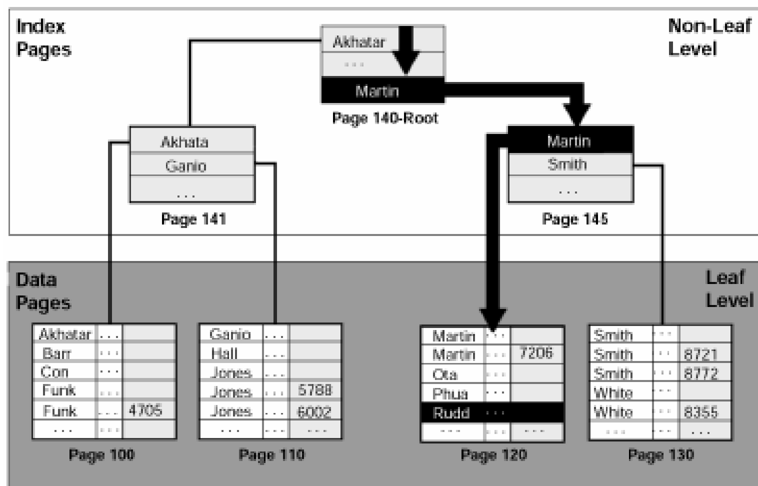


Figure 3: Query on a Clustered Table[10]

Non-clustered indexes work in much the same way except that the leaf level holds a pointer rather than the actual data. This pointer is the row ID so that the data can be retrieved. However, this lookup can add great expense to a search because the result data is not physically next to each other, so data may need to be read from all over the file. Figure 4 shows the same search in Figure 3, but on a heap table.

```
SELECT lastname, firstname FROM member WHERE lastname = 'Rudd'
```

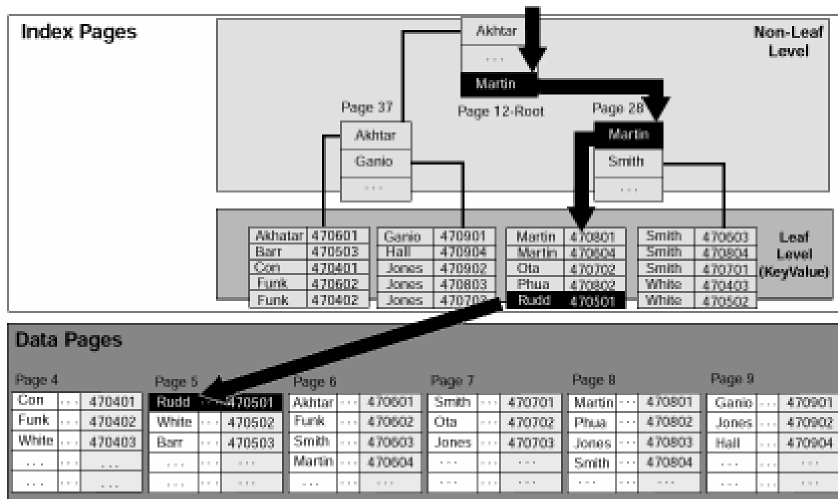


Figure 4: Non-clustered Index on a Heap[10]

Non-clustered indexes on a clustered table work much the same way with an additional step. Rather than a row ID, the pointer contains clustered index key values and SQL Server obtains a list of all these keys to perform a new lookup. These lookups are very fast, but they must be done multiple times. Since these pointers are not contiguous, each record must be looked up individually on the clustered index. Figure 5 shows a clustered table with a non-clustered index.

```
SELECT lastname, firstname FROM member WHERE member_no = 6078
```

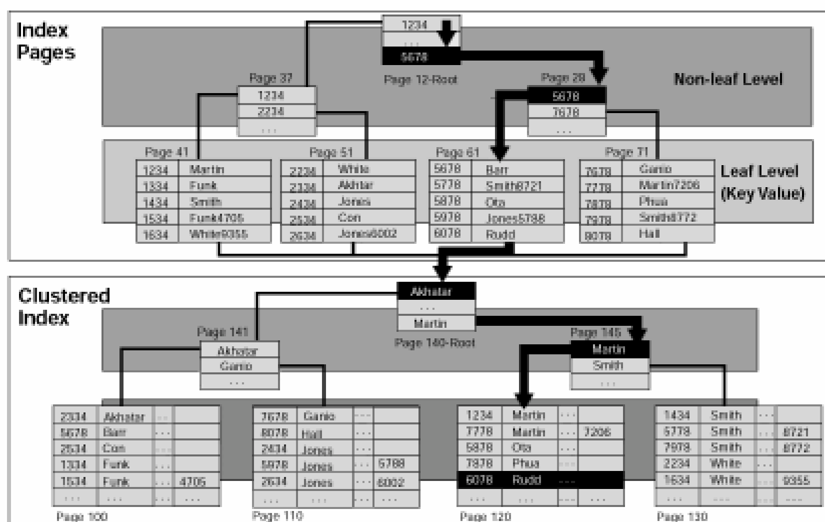


Figure 5: Clustered Table with a Non-clustered Index[10]

In this work we control the effect of indexing by having two indexing scenarios for each experiment

3.5 Data Distribution

The contents of the database can have a large impact on query performance. SQL Server maintains a variety of statistics to monitor data distribution, and enlists them to determine the best plan of action for query execution. SQL Server records the number

of rows in a table, the number of pages occupied by the index or table, the time the statistics were collected, and a string summary if the column contains character data. SQL Server keeps a histogram of the table data to examine data distribution. The histogram is used to determine the selectivity of an index. Selectivity is the percentage of values in a column that are unique. This is crucial for non-clustered indexes because each lookup in a non-clustered index could result in multiple additional lookups in a non-clustered index. For example, an index on a column with low selectivity such as a last name field may be grossly inefficient for a query such as

```
Select c.firstName, c.lastName
from Person.Contact c
WHERE c.LastName = 'SMITH'
```

The low selectivity of the index makes a table scan more efficient, and in general, an indexed column should have 90-95 percent uniqueness[10] to justify the expense of the additional lookups generated by the non-clustered index. Because data changes over time, the statistics should be updated as well to ensure the statistics correctly portray the data in order to make the most accurate assumptions as to which indexes to use or not use.

Data distribution is another variable that must be controlled to ensure experimental integrity. The DBCC SHOW STATISTICS command[5] will list all pertinent statistics to verify distribution. In this work, two data distributions are used to control the effect that data distribution can have on SQL phrasing performance. The two data distributions are specified by the TPC-C specification, which we implement for our experiments.

3.6 Query Optimizer

The query optimizer takes the gathered statistics and generates a query plan to balance fast results and minimal impact on other users. When the database is queried, the SQL Parser checks for proper syntax and converts the query to a relational algebraic expression. The Query Optimizer then makes a cost-based analysis to choose an execution path taking into account available indexes and data statistics. SQL Server Management Studio offers a graphical representation of both the estimated execution plan and actual execution plan. The two may differ dramatically as the estimate execution plan does not run the query and is based solely on statistics. If there are outdated or no statistics available for columns, the estimate may be inaccurate. The actual execution plan is shown after a query is completed. Figure 6 is the actual execution plan from a query executed on the AdventureWorks Sample database.

```
SELECT c.firstName, c.lastName
FROM Person.Contact c
INNER JOIN HumanResources.Employee e
ON c.contactid=e.contactID
WHERE c.LastName LIKE 'S%'
```

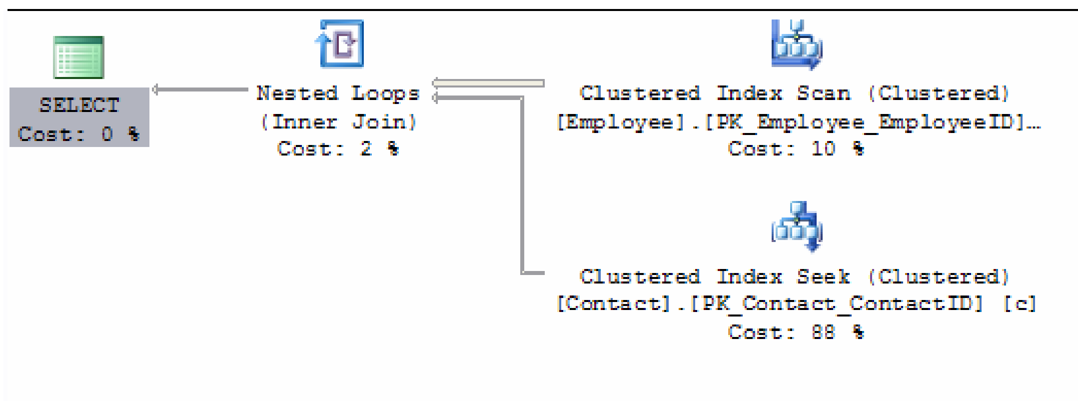


Figure 6: Execution Plan

We will inspect query plans of the tested phrasings to verify that they are executing differently, and the optimizer does not convert them to run the same plan. We will also review them after the queries execute to infer performance differences.

SECTION 4: EXPERT GUIDANCE

To gain insight into what groups may benefit from researching SQL phrasing, I visited a few local database administrators to learn more about their data environments and the day-to-day effects of SQL phrasing. We discovered the size and nature of an organization greatly impacts the perception of SQL phrasing.

4.1 Vendor Dependent Shops

Large organizations often opt to purchase prepackaged solutions to meet their data needs. This can be less expensive than developing a custom solution and has been through rigorous testing to ensure proper performance. New Hanover Regional Medical Hospital and the University of North Carolina at Wilmington both selected vendor backed enterprise solutions. These systems have been configured to work as installed, and their user licenses dictate that changes must be made with cooperation of the vendor. The database administrators (DBA) monitor the system and report performance issues to the support team. DBAs examine CPU loads, network traffic, and unreleased locks to identify problems and then notify the vendors for a resolution. Through monitoring, administrators see the impact of suboptimal queries, noting that rephrased queries have pruned minutes off query execution times. Our research is

unlikely to assist DBAs inside monolithic organizations as they are limited by their license agreements to alter SQL phrasings.

4.2 Custom Deployment

In contrast, organizations whose business advantage lies within the database space can greatly benefit by concentrating on SQL wording. Vendors and agencies that optimize their phrasings will achieve better overall performance than a competitor who does not.

Castle Branch Incorporated is an investigative screening company that sets itself apart from its rivals with a custom made customer relations management system. Castle Branch is committed to honing their SQL phrasing and has a DBA in charge of improving execution times. A performance threshold is set for execution times and when a query consistently exceeds the limit, an email notification is sent to the DBA. The DBA then runs the query through a custom scripted performance evaluator to verify that it requires enhancement. She looks for performance impediments as well as ways to improve the query and runs different phrasings through the evaluator to find the optimal solution. She points out that SQL phrasings are often written by application developers whose primary focus is on making a system operational, and that leaves room for improvement. My research is ideal for companies such as Castle Branch who have a vested interest in the performance of their queries as well as a means to alter them.

SECTION 5: EXPERIMENTS

5.1 Assumptions

For our experiments to have value to database administrators, the trials will occur on a system that has properties consistent with an online transaction processing environment. This includes creating appropriate levels of fragmentation and proper consideration to cache memory. These assumptions will be discussed further in their own sections.

5.2 Setup of Environment

5.2.1 Server

As part of this project, a custom database server was built during the Fall of 2008 and the configuration is listed in Figure 7.

COMPONENT	SPECIFICATION
Operating System	Microsoft Windows 2003 Server Edition 64-bit
Drives	C: 150 GB RAID 1
	D: 150 GB RAID 1
	E: 450 GB RAID 10
CPU	Intel Xeon 3.0 GHz
Memory	8GB
Database Management System	SQL Server 2008 64-bit

Figure 7: Test Server Description

Drive C holds the operating system and SQL Server. Drive E holds the database files and Drive D holds the log files. This is Microsoft's recommendation[3] given the number of drives available to ensure data reliability and speed.

5.2.2 Data Model

Several options were weighed in selecting an appropriate data model. We could have built a custom design that would meet all the needs of the trials. However, this adds additional time to the project. Testing and justification would need to be documented, and the design decisions would be scrutinized as much as the experiments themselves. We chose instead to select from a number of well-defined database designs. This enables the focus to be solely on the experiments, and practitioners would already be familiar with the data model. Initially, Microsoft sample databases Adventureworks and Northwinds were top candidates as they are well known. They were created, however, only as training databases and are more extensive designs than we require. The choice candidate was the data model described in TPC Benchmark C[9].

The Transaction Processing Performance Council is an organization that tests hardware and database management systems using a standardized data model as a control. TPC's benchmarks have become the de facto standard for comparing server hardware and DBMS. IBM, Dell, Oracle, Sun, & Microsoft all subscribe to the TPC and their process is held in high regard within the database community. Their data models have been rigorously tested, are well documented, and are made for transaction processing environment making them ideal candidates for our experimentation. The model as

described by TPC Benchmark C was chosen as it provides a method for scaling, is easy to grasp, and has extensive documentation to complete the data model within SQL Server.

5.2.3 TPC Benchmark C Data Model

The data model represents a company that is a wholesale supplier with geographically distributed sales districts and associated warehouses. Each regional warehouse covers 10 districts and each district serves 3000 customers. Customers order from the company's 100,000 items available. The model is scalable by the number of warehouses as seen in Figure 8 below.

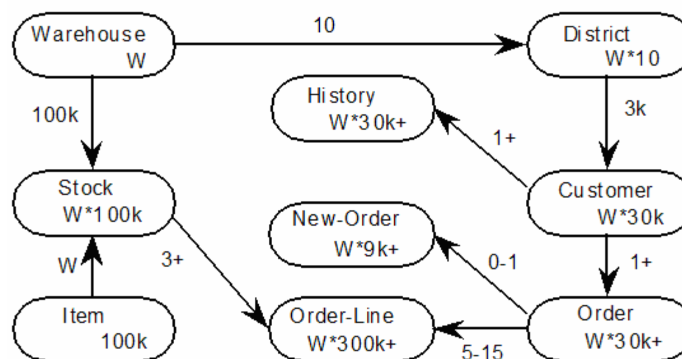


Figure 8: TPC Data Model

5.2.4 Implementation

The data model was constructed as described in pages 11-18 of the TPC Benchmark C. TPC-C documents primary keys, but makes no mention of indexes. For the

experiments, we shall use clustered indexing on all primary keys as the minimal index set as shown in Figure 9. Figure 10 shows the additional indexes used in the extensive indexing queries, where an index on the item number and order date fields will be

TABLE	INDEX	INDEX TYPE
dbo.WAREHOUSE	W_ID	CLUSTERED
dbo.DISTRICT	D_W_ID, D_ID	CLUSTERED
dbo.CUSTOMER	C_W_ID, D_D_ID	CLUSTERED
dbo.ITEM	I_ID	CLUSTERED
dbo.STOCK	S_W_ID, S_I_ID	CLUSTERED
dbo.ORDERS	O_ID, O_W_ID, and O_D_ID.	CLUSTERED

added which should have an impact on the queries in Appendix A.

Figure 9: Minimal Index Set

TABLE	INDEX	INDEX TYPE
dbo.ORDERLINE	O_I_ID (item number)	NONCLUSTERED
dbo.ORDERS	O_ENTRY_D	NONCLUSTERED

Figure 10: Extensive Index Set (in addition to Minimal Index Set)

The TPC documentation gives explicit instruction on populating the tables. For each table, there is a stored procedure that follows the TPC instructions to fill the tables. There are also stored procedures to produce the required field values such as zip codes, random strings, and phone numbers. The following shows the procedure to populate the WAREHOUSE table:

```
CREATE PROCEDURE [dbo].[uspPopulateWarehouse]
        @numWarehouses int
        AS
        DELETE  FROM dbo.WAREHOUSE
        DECLARE @charsetmin int = 32
        DECLARE @charsetmax int = 160
        DECLARE @i int
        SET @i = 0
        WHILE @i < @numWarehouses
        BEGIN
                SET @i = @i+1
```

```
            declare @name varchar(10)
            exec dbo.uspRandomString 6, 10, @charsetmin, @charsetmax, @Str = @name OUTPUT
            declare @street1 varchar(20)
            exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @street1 OUTPUT
            declare @street2 varchar(20)
            exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @street2 OUTPUT
            declare @city varchar(20)
            exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @city OUTPUT
            declare @state varchar(2)
            exec dbo.uspRandomString 2, 2, @charsetmin, @charsetmax, @Str = @state OUTPUT
            declare @zip varchar(9)
            exec dbo.uspMakeZip '11111', @zip = @zip OUTPUT

      declare @tax numeric(4,4)
            exec dbo.uspMakePercentage 0.2, @percentage = @tax OUTPUT

            INSERT INTO dbo.WAREHOUSE
            (W_ID, W_NAME, W_STREET_1, W_STREET_2, W_CITY, W_STATE, W_ZIP, W_TAX, W_YTD)
            VALUES
            (@i, @name, @street1, @street2, @city, @state, @zip, @tax,300000)
    END
    RETURN
```

Populating the database requires substantial random data generation. To create a replicable dataset, we will seed the rand() function and this will be listed within the respective stored procedures.

5.2.5 Data Scenarios

The base scenario for each of the query trials are the indexes described above with uniformly distributed data. Our experiments will expand these data scenarios to examine the effects of data distribution and indexing on query performance. As noted previously, indexing can have a profound impact on execution times, and different phrasings may make better use of them than others. Data distribution should also be explored. We would like to see how selectivity impacts performance and if there is a discernable difference between uniform and nonuniform distribution. The TPC documentation outlines a function for both distributions and they have been rendered via stored procedures. Each experiment will run through the performance matrix in Figure 11.

Query Phrasing	Minimal Indexing, Evenly Distributed Data	Minimal Indexing, Nonuniform Data Distribution	Extensive Indexing, Evenly Distributed Data	Extensive Indexing, Nonuniform Data Distribution
Phrasing				
Alternative Phrasing				

Figure 11: Experiment Matrix

5.2.6 Metrics

The primary basis for query comparison is elapsed execution time. This is measured by a stored procedure that takes a timestamp directly before and after a query. The difference is computed to obtain the run time. This method eliminates any overhead associated with the iterations. We will also examine the execution plans and SQL Server statistics to gain additional insight into performance differences. Reviewing statistics such as the I/O time, CPU time, number of logical reads, and number of physical reads can reveal why one query is faster than another. The execution plans will verify that the queries executed differently and make clear the reasons for any performance disparity.

5.3 Experiment: Subquery vs. Join

A common question that arises in SQL phrasings is when to use joins and where subqueries are more appropriate. Joins match one record from one table up with one or

more records from another table to produce a set with the combined selected columns from both tables. A subquery is a T-SQL query that is nested within another query to serve as part of the data or a condition in another query. Most texts suggest that the multi-query aspect of the subquery makes them slower than an equivalent join statement. In our experiment, we will draw on a subquery and an equivalent join to produce a record set containing columns from dbo.ORDER-LINE, dbo.ORDERS, and dbo.CUSTOMER. Hypothesis 1 is that the join statement will be faster in all tested scenarios. I am concerned, however that I have been unable to find any data to support the best practice claims. My goal is that our research will provide practitioners with quantitative evidence to support their phrasing choices.

JOIN
```
SELECT DISTINCT O_C_ID, O_ID
        FROM ORDERS  JOIN
                ORDERLINE
                ON O_ID = OL_O_ID
                WHERE OL_I_ID = @itemid
```

SUBQUERY
```
SELECT DISTINCT O_C_ID,O_ID
                FROM ORDERS
                WHERE O_ID
                  IN (SELECT OL_O_ID FROM ORDERLINE WHERE OL_I_ID = @itemid )
```

Figure 12: Subquery vs. Join Queries

5.4 Experiment: Date Comparisons

SQL Server provides a variety of date comparison strategies, and we will explore the performance implications in choosing between two common approaches. The business situation we will be investigating is returning sales order data within a specified date range. The first method will take advantage of the BETWEEN construct to return the results, and the other query will use Year(DATE). As stated in Hypothesis 2, Date Between will be faster than Date Part in all tested scenarios. Hypothesis 2 was

established because Date Between does not have to use additional time to parse the date as is the case of Date Part. This trial has the potential to have vastly disproportionate times with extensive indexing, as Date Between has the potential make more efficient use of an index on *O_Entry_Date* than would Date Part.

```
DATE BETWEEN
SELECT O_C_ID,O_ID, O_ENTRY_D
                FROM ORDERS
                WHERE O_ENTRY_D BETWEEN @rangestart AND @rangeend

DATEPART
SELECT O_C_ID,O_ID, O_ENTRY_D
                FROM ORDERS
                WHERE DATEPART(M, O_ENTRY_D) = @datemonth AND
                    DATEPART(year, O_ENTRY_D) = @dateyear
```

Figure 13: DATE PART vs. DATE BETWEEN Queries

5.5 Experiment: MERGE

Other database products, notably Oracle, have had a merge construct for over a half decade. This merge construct is introduced to SQL Server in the 2008 release[4]. A merge allows a user to conditionally update or insert data from a table into a target table. This is particularly useful for data migration, and previously SQL Server users had to create their own "upsert" procedure. This is a hand coded solution that, through a series of loops and inserts, performs identical functionality to MERGE. As it is a user-defined construct, there are many variations on upsert methods, making troubleshooting and maintenance difficult. The MERGE statement is a SQL Server-defined construct which is well documented, and its results are easily replicated from system to system. Our experiment will consider a scenario of an expanding business that is adding products from a recently acquired competitor. The goal is to unite two product catalog

tables. We will take a table of 40,000 products (the acquired competitor's database)- half of which are included in the dbo.Item table and half of these are not (the acquiring company currently sells half the products of the overtaken organization), and perform both a MERGE and an upsert procedure that we have created. It is significant that there be a mixture of items that need inserting and those that do not. Hypothesis 3 states that Merge will be faster in all tested scenarios. See Appendix A for the tested Merge and Upsert queries.

SECTION 6: EXPERIMENT REVISIONS

Sections 1-4 explained the background and planned execution of a SQL Server performance experiment. The experiment was presented to the defense committee and there were suggested alterations to improve the trials.

6.1 Experiment Revisions: Size as a variable

At the advice of the defense committee, the data scenario in Section 4.2.5 has been expanded to include database size measured in number of rows as a variable. There will be three Figure 10s : one each for a small, medium, and large database.

Size	Minimal Indexing, Evenly Distributed Data	Minimal Indexing, Nonuniform Data Distribution	Extensive Indexing, Evenly Distributed Data	Extensive Indexing, Nonuniform Data Distribution
Small	Phrasing A, Phrasing B	Phrasing A, Phrasing B	Phrasing A, Phrasing B	Phrasing A, Phrasing B
Medium	Phrasing A, Phrasing B	Phrasing A, Phrasing B	Phrasing A, Phrasing B	Phrasing A, Phrasing B
Large	Phrasing A, Phrasing B	Phrasing A, Phrasing B	Phrasing A, Phrasing B	Phrasing A, Phrasing B

Figure 11 Revised

6.2 Experiment Revisions: TPC Setup

The TPC ratios between warehouses, customers, orders, and order lines are not amenable to scaling. A revised and simplified data model is shown in Figure 11. This model generates 100000 orders and approximately 1 million order lines for each warehouse. The HISTORY, NEW-ORDER, and STOCK tables from Figure 8 were omitted as they were unnecessary for the trials, and the processing time necessary to generate data for the three tables would be significant.

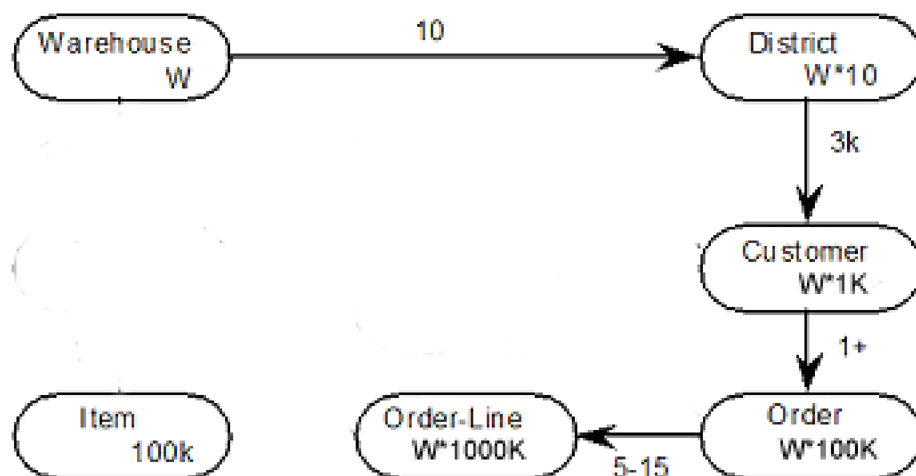


Figure 14: Fig. 8 Revision

6.3 Experiment Revisions: MERGE

The scenario described in 4.3 is not conducive to scaling in the TPC test environment. The tested scenario is merging the ORDERS table into a data warehouse table. Approximately 40% of the records in the ORDERS table will be inserted into the data warehouse with the remaining records updated in the data warehouse.

SECTION 7: EXPERIMENT EXECUTION

This section will describe the data generation, data collection, and specific queries test.

7.1 TPC Database

A single TPC base database containing all necessary tables, views, and procedures for executing the trials and storing data was created as shown in Figure 12. Using SQL Server's *Copy Database* feature, twelve different database permutations (3 sizes (small-medium-large)* 2 indexing schemes (minimal-extensive)*2 distributions(uniform-nonuniform) were created and added to the server to in order to complete the trials. Before creation, the disk was defragmented in an effort to eliminate fragmentation. As discussed in the proposal, fragmentation can alter seek times and is an untested variable. Defragmenting, along with set index fill factors, should keep fragmentation's effect to a minimum.

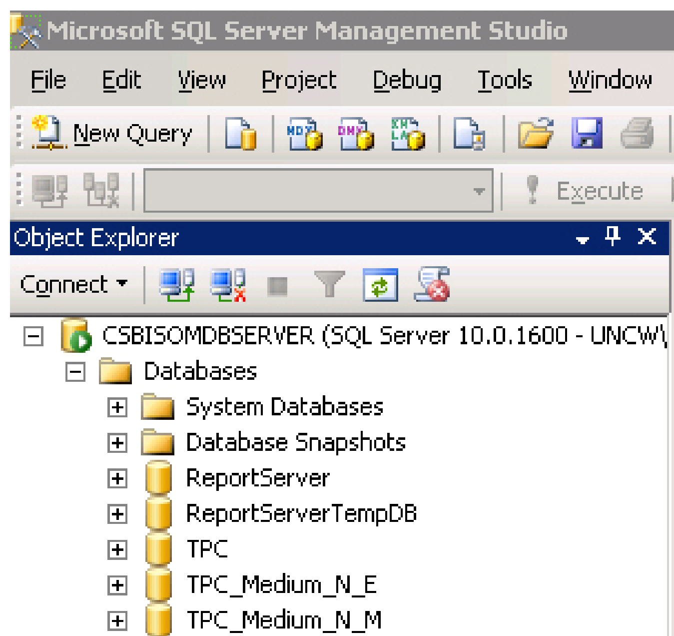


Figure 15: TPC base DB and 2 copies for experimentation

Each of the databases was populated with a stored procedure call containing the variables appropriate for its specific permutation.

7.2 Test Execution

The tested queries in Appendix A are stored in their respective stored procedures which allow execution statistics to be captured with dynamic management views[6]. Each experiment is run via a stored procedure that calls the compared queries for each row in the sample set. The cache is cleared preceding and following each query to ensure accurate results. The experiments were ran via SQL Server Agent jobs, and each procedure was ran individually without any other nonessential processes running. The results were output to three tables within each database, and later compiled into a separate database for analysis.

SECTION 8: Data Analysis

8.1: Collection and Analysis. For each observation, the same task was given to each method and was performed on the same data. Each data row in the results table was paired with its alternate method partner and the difference of the dmv execution time was taken.

Results in SQL Server

	StudyType	Size	Indexing	Distribution	runID	RunType	ItemId	numRecords	StopWatchTime	dmvTime
1	Join	Small	Minimal	Uniform	1	JOIN	23761	6	657	653
2	Join	Small	Minimal	Uniform	1	SUBQUERY	23761	6	1280	1277
3	Join	Small	Minimal	Uniform	2	JOIN	63392	12	1237	1239
4	Join	Small	Minimal	Uniform	2	SUBQUERY	63392	12	1347	1332
5	Join	Small	Minimal	Uniform	3	JOIN	10437	11	1233	1231
6	Join	Small	Minimal	Uniform	3	SUBQUERY	10437	11	1233	1231

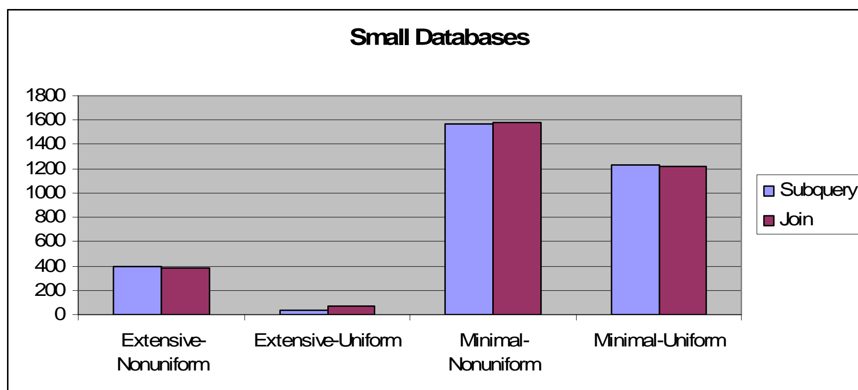
Difference of Query Times Prepared for Analysis

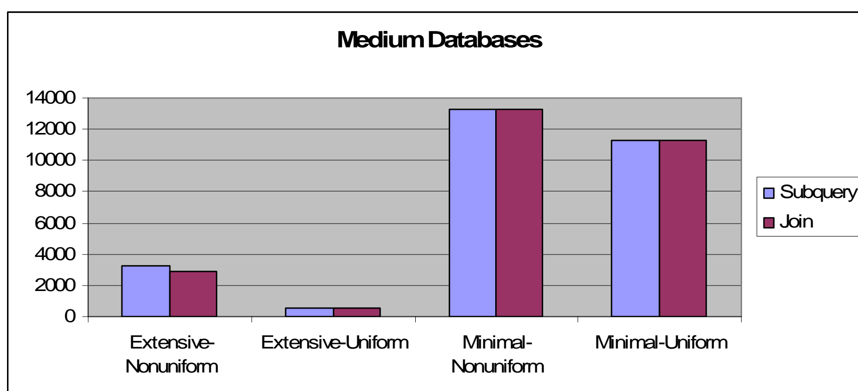
	A	B	C	D	E	F	G	H
1		Subquery - Join Time						
2								
3	SIZE			DISTRIBUTION			INDEXING	
4	Small	0		Uniform	0		Minimal	0
5	Medium	1		NonUniforr	1		Extensive	1
6	Large	2						
7								
8	size	Indexing	Distribution	Difference	runID			
9	0	0	0	624	1			
10	0	0	0	93	2			
11	0	0	0	0	3			

Figure 16: Results Prepared for Analysis

A multivariant analysis of variance (MANOVA) was performed on the difference results to determine the statistical significance of the size, indexing, and distribution variables. The mean difference of each trial was examined to determine the best performing query option with the hypothesis preferred phrasing time being subtracted from the alternative phrasing time i.e. diff = time(subquery) – time(join).

8.2 Join Versus Subquery: Hypothesis 1 was that join would be faster on average than the equivalent subquery phrasing in all scenarios, and was true for 8 of the 12 tested scenarios. However, in only 3 instances was the difference greater than a 3% performance difference, including the most dramatic relative performance increase in favor of the subquery. In the small size-extensive indexing-uniform distribution trial, the average subquery trial was 30ms, while the join phrasing took on average 70ms to complete. It did take twice as long, but in practice, the 40ms differential is negligible.





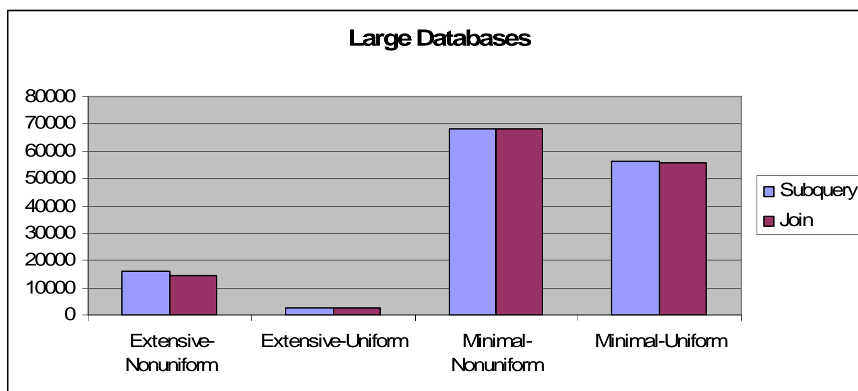


Figure 17: Subquery vs. Join Average Runtime Comparison (in milliseconds)

The other noticeable differences favored the join query. In the medium and large size-extensive indexing-nonuniform distribution trials, join query times (2931ms and 14704ms respectively) outperformed subquery (3287ms and 16026ms) for performance gains of 8.25% and 10.85%. This translates to gains in the medium trial of ¼ of a second and 1.25 seconds in the large trial which could amount to significant time savings if the queries were executed often. The complete results of the Join/Subquery trial are listed in Appendix B and Figure 15 shows the side-by-side comparison of each trial's average times.

According to the MANOVA, all variables contributed to the difference between the two queries. Size was the largest factor, and this is attributed to the large difference in the Large-Uniform-Minimal trial and the Large-Extensive-Nonuniform (L-E-N) trial. In the case of the L-E-N trial, the greater selectivity of the index may account for the increased performance of the Join query. This is supported by the large impact of size*indexing*distribution and indexing*distribution variables in the MANOVA. In the medium database trial, the join phrasing clearly makes a difference given extensive indexing on nonuniform data (2931ms vs. 3287ms). The query plans are virtually identical for the small and medium databases, and frequently the subquery was faster than the join (denoted by the negative average mean), which would seem to reject Hypothesis 1. However, when the query plans differed, it was often an increase in parallelism on the part of the subquery. SQL Server 2008 employs parallelism when the number of processors exceeds the number of active connections and/or when the estimated execution plan for the serial execution of the query is higher than the query plan threshold. The result is one query enlisting more processing power than its

counterpart, and while it may have more total CPU time, its elapsed runtime will be less. This is a variable that was previously unconsidered, and is generally controlled via query analyzer. SQL Server 2008 does allow for user management of parallelism, and can be manipulated in subsequent trials. Data suggests that query phrasing contributes to run times in the cases of medium and large databases with nonuniform data and extensive indexing. Therefore, practitioners should always opt for a join phrasing when presented with similar data settings as there is no compelling evidence to favor subqueries in any of the situations.

8.3 Date: Hypothesis 2 stated that DATE BETWEEN yield faster run times on average in all scenarios. The data indicates that it is faster in 10 of the 12 tested situations, with significant differences in all 6 extensive indexing scenarios. The two cases where DATE PART was the faster phrasing on average, small and medium sized databases with nonuniform data and minimal indexing, did not provide a

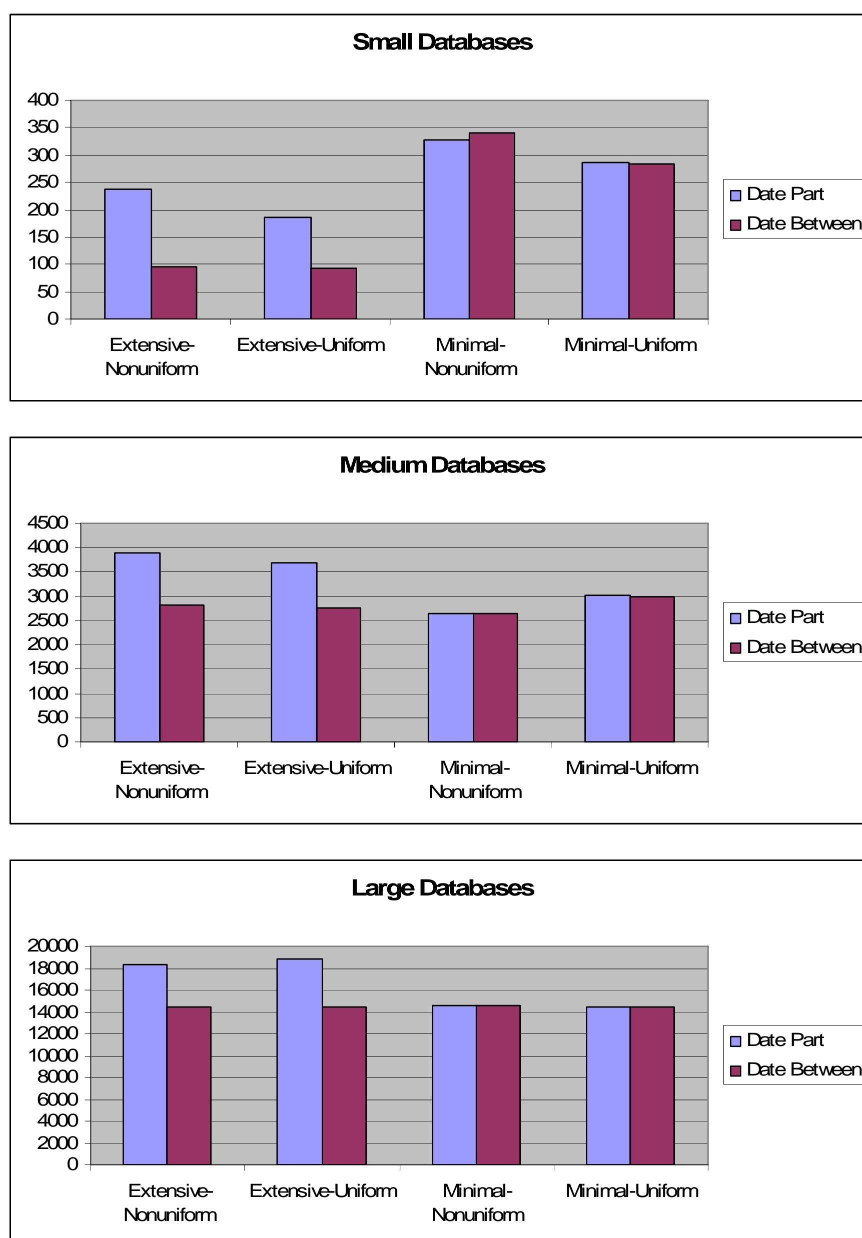


Figure 18: Date Comparison of Average Run Times (in milliseconds)

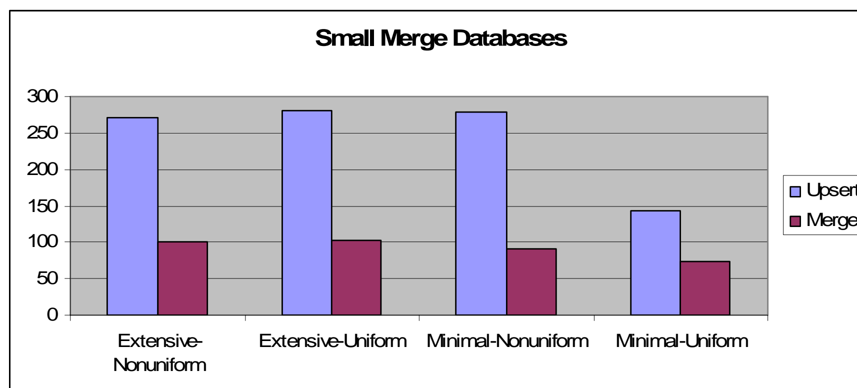
significant difference. In the small case, it was 329ms versus DATE BETWEEN's 341ms and just 1ms difference in the medium trials. Furthermore, the query phrasing did not make a practical difference in the execution times with no more than a 3% difference in any minimal indexing trial. The added index on O_ENTRY_D had a drastic improvement on the execution time of the DATE BETWEEN query according to the MANOVA indicating the significance of the indexing and size*indexing variables. Examining the average differences tables, DATE BETWEEN is faster in all extensive indexing cases with 50% improvement in small trials, over 25% improvement in medium sized databases, and over 20% gains in large trials. This translates to a 4 second improvement by utilizing DATE BETWEEN rather than DATE PART (14395ms vs. 18343ms for Large-Extensive-Nonuniform and 14463ms vs. 18827ms for Large-Extensive-Uniform). Figure 16 shows the side-by-side comparison of each trial's average times and Appendix C contains the full statistical information. It is important to note that the index, while present, is not necessarily used. Appendix E shows the execution plans of the DATE BETWEEN query on the Medium-Uniform-Minimal Indexing (top) and Medium-Uniform-Extensive (bottom) with runtimes of 900ms and 3026ms, respectively. They are identical with the query analyzer making a recommendation for an index *ON* O_ENTRY_ID that *INCLUDES* O_ID and O_C_ID. As explained in *2.4*, the leaf nodes of the nonclustered index contain a pointer to the physical location, and the expense of the index seek and pointer lookup can be expensive. The benefit of the index is a measure of the selectivity of the data.

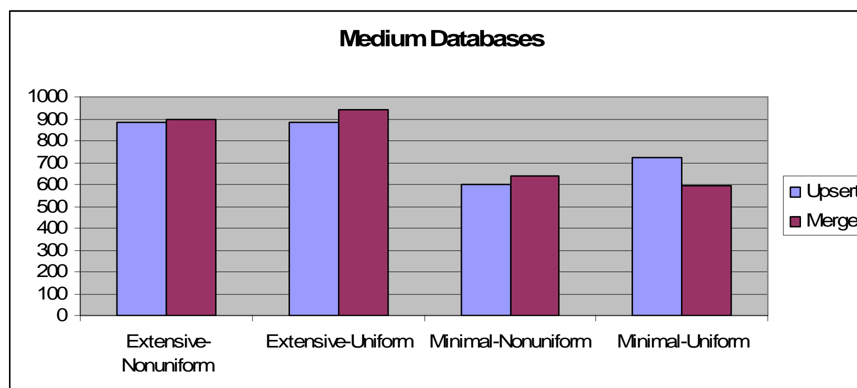
The query analyzer determined that it was more efficient to scan the clustered index (clustered indices in the experiment are the primary key values) rather than perform the

index seek. The query analyzer's suggestion to use the *INCLUDES* clause would induce the query analyzer to use the additional index. Columns in the *INCLUDES* clause are added only at the leaf level. SQL Server will stop looking for information once it has everything it requires, and if it finds all necessary information in the leaf level of the index, it does not need to follow a pointer to the physical location. This unburdens the system from the additional I/O related to the lookups, but there is a price. The additional information at the leaves causes fewer rows to fit on an index page, and more I/O will be needed to see the same number of rows. This may cause one query to perform faster while slowing down others. Identical queries executed on two databases containing identical data with a statistically significant difference in execution times suggest that the physical location has a substantial impact on overall performance. This calls into question the results of the study, and ensuing studies should aim to control the variable. For practitioners, DATE BETWEEN should be used in conjuction with indexing on the date column. Without indexing, it makes little difference how the query is phrased.

8.4 Merge: Hypothesis 3 stated that all merge would be faster on average in all situations. The results revealed that it was faster in 8 of the 12 scenarios, but was slower in the medium scenarios with the exception of uniform data with minimal indexing and the large database trial with uniform data and minimal indexing. Appendix D contains the analytical results of the merge/upsert trial and Figure 17 shows the average runtimes of each query. Merge was the favored phrasing in all small database instances with average times near 100ms in contrast to average Upsert times

near 250ms. For practitioners, this is not significant. For medium sized databases, merge was on average 3% worse in every case except uniform data with minimal indexing where it performed better(592ms for Merge vs. 725ms for Upsert). For large databases, the greatest benefit of using the merge phrasing came with nonuniform indexing where it enjoyed a 355ms advantange on average for a near 8.5% performance improvement. For practitioners, this suggests that merge is preferable in small databases and large databases with nonuniform distribution. For medium databases, it may be advantageous to create a custom upsert to perform the operation.





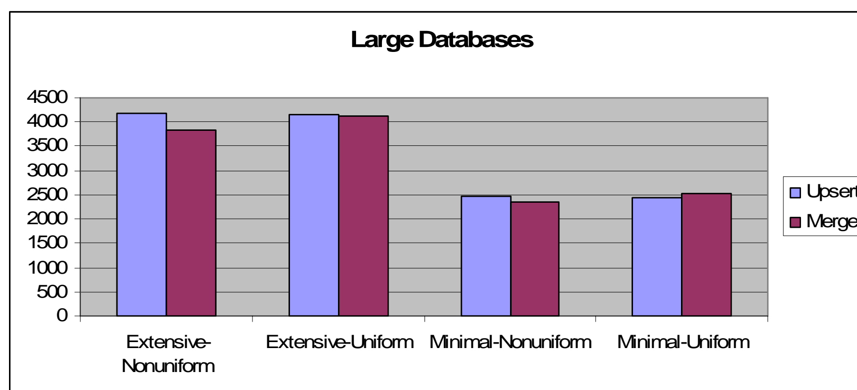


Figure 19:Merge vs. Upsert Comparison of Average Runtimes (in milliseconds)

The MANOVA suggested the majority of the findings were attributed to factors other than size, distribution, and indexing. The varied average run times are also suspect as the query plan of the merge statement includes the creation of at least one temp table, management of indexes, and management steps that are expensive. One would expect a trend of larger comparative differences between the queries as the overhead relative to total execution time decreases, but the run times reveal just the opposite. The merge statement is worthy of additional consideration.

SECTION 9: SQL Server Management Studio Review

Without the features of SQL Server Management Studio, the trials would have been much more difficult to execute. Dynamic Management Views (DMVs) allow for a look inside the internal operation of SQL Server to gather information without the need to search in system tables. DMVs saved time in experiment design as there was no need to decipher system tables, and the SQL Server API provided the details of the proper DMV to utilize. The graphical user interface is intuitive and can be educational. The GUI allows the user to perform scriptable operations such as create objects and indexes. Through the *Script As* option, SQL Server will output the commands as script, allowing the user to see how to perform an operation without the GUI. The Object Explorer allows for easy navigation of objects with a clear separation of tables, views, and functions. The *Copy* task using SQL Server Integration Services made it convenient to create the 12 data scenarios as their own databases. The most helpful aspect of SSMS was the improved IntelliSense. IntelliSense is the editor feature that color codes comments and key words, and underlines syntactical scripting errors. It will

also autocomplete the names of constructs as you type. These features saved a great deal of time in typing as well as error detection and correction. Finally, the ability to view the execution plans allowed a greater insight into the tested phrasings to verify how SQL Server chose to carry out the query. This would be a true asset in query tuning.

SECTION 10: Future Research

The Merge construct deserves further investigation due to the MANOVA results. Given that it is a new construct for SQL Server 2008, it is open to additional research and collaboration can expound the knowledge base. The trials set lower bounds for management of system resources, but did not take into account upper bounds, which may have resulted in unforeseen introduction of variables. One query was free to receive more memory or CPU time than another at the discretion of the Query Analyzer. Further studies should seek to control parallel processing or capture its impact by measuring CPU time if possible. Rather than use a different database for each indexing scheme, subsequent studies should take advantage of the *DISABLE* indexing option, which will render the index unusable until it is *REBUILT*. This approach allows the research to use only 6 databases (one for each size and distribution type) and toggle the additional indexes on or off to control minimal or extensive indexing. A variable that was kept constant during the trials was the fragmentation level, with a fill factor of all indexes set to 20% free space on pages. Query performance could vary significantly if the fragmentation level was varied. For instance, higher fragmentation would result in fewer records fitting on a data page, and performance would likely suffer. In contrast,

100% space utilization would likely see an improvement SELECT statements, but INSERT statements may take longer as there is a greater likelihood of page splits. To consider fragmentation as a variable, the trials would need to include deletes and inserts throughout a record set to model a production environment in order to see the effect that fragmentation has on the execution times. A final note should be made about the performance effect of caching. If a page is in cache, the I/O required to retrieve the page from disk is bypassed and is a boon to performance. The trials cleared the cache before each query run, and thus is a worst case run time. The size of cache and the data in cache at any one time are other variables that may be considered in comparing query times. The size of cache affects the number of pages in cache at any one time, and a larger cache increases the likelihood that the query will need I/O to go to disk. If the needed pages are almost always in cache as is often the case with repeated queries, the execution times would likely be faster.

References

1. Ben-Gan, Itzik (2008, July) Introduction to New T-SQL Programmability Features in SQL Server 2008 White Paper. (http://technet.microsoft.com/en-us/library/cc721270.aspx)

2. Ben-Gan, Itzik, Lubar Kollar, Dejan Sarka (2006). Inside Microsoft 2005: T-SQL Querying. Microsoft Press.

3. Database Performance. SQL Server 2008 Books Online http://msdn.microsoft.com/en-us/library/ms190619.aspx

4. Delaney, Kalen (2008). Inside Microsoft SQL Server 2005: Query Tuning and Optimization. Microsoft Press.

5. Larsen, Greg (2007, December). Measure TSQL Statement Performance. http://www.sql-server-performance.com/articles/per/tsql_statement_performance_p4.aspx

6. Larsen, Greg (2008, January). Query Execution Statistics. (http://www.sql-server-performance.com/articles/per/Query_Execution_Statistics_p1.aspx)

7. Sledge, Orryn, Mark Spenik (2002, November). Microsoft® SQL Server™ 2000 DBA Survival Guide, Second Edition. Sam's Publishing.

8. SQL Server 2008 Books Online (2008, October). Understanding Pages and Extents. (http://technet.microsoft.com/en-us/library/cc721270.aspx)

9. Transaction Processing Performance Council (http://www.tpc.org)

10. Vieira, Robert (2007). Professional SQL Server 2005 Programming. Wiley Publishing.

Hypothesis 1: The Join statement will be faster than the Subquery statement in all tested scenarios.

JOIN

```
SELECT DISTINCT O_C_ID, O_ID
        FROM ORDERS  JOIN
                ORDERLINE
                ON O_ID = OL_O_ID
                WHERE OL_I_ID = @itemid
```

SUBQUERY

```
SELECT DISTINCT O_C_ID,O_ID
                FROM ORDERS
                WHERE O_ID
                  IN (SELECT OL_O_ID FROM ORDERLINE WHERE OL_I_ID = @itemid )
```

Hypothesis 2: The Date Between statement will be faster than the Date Part statement in all tested scenarios.

DATE BETWEEN

```
SELECT O_C_ID,O_ID, O_ENTRY_D
                FROM ORDERS
                WHERE O_ENTRY_D BETWEEN @rangestart AND @rangeend
```

DATEPART

```
SELECT O_C_ID,O_ID, O_ENTRY_D
                FROM ORDERS
                WHERE DATEPART(M, O_ENTRY_D) = @datemonth AND
                    DATEPART(year, O_ENTRY_D) = @dateyear
```

Hypothesis 3: The Merge Statement will be faster than the Upsert query in all tested scenarios.

MERGE

```
MERGE ORDERS_DATAWAREHOUSE as TARGET
        USING dbo._mergesource AS SOURCE ON
        (TARGET.O_ID = SOURCE.O_ID AND TARGET.O_D_ID = SOURCE.O_D_ID AND
TARGET.O_W_ID = SOURCE.O_W_ID)
          --If the record is not in the data warehouse, insert it
                WHEN NOT MATCHED THEN INSERT VALUES (SOURCE.O_ID, SOURCE.O_D_ID,
SOURCE.O_W_ID, SOURCE.O_C_ID, SOURCE.O_ENTRY_D, SOURCE.O_CARRIER_ID,
SOURCE.O_OL_CNT, SOURCE.O_ALL_LOCAL)
--If the record is in the data warehouse with ALL_LOCAL variable equal to 9, update it with the source
record
          WHEN MATCHED AND TARGET.O_ALL_LOCAL = 9 THEN UPDATE SET
TARGET.O_ALL_LOCAL = SOURCE.O_ALL_LOCAL
        ; --MERGE statements must end with a ';'
```

UPSERT

```
UPDATE ORDERS_DATAWAREHOUSE
 SET  ORDERS_DATAWAREHOUSE.O_ALL_LOCAL = T1.O_ALL_LOCAL
 FROM  ORDERS_DATAWAREHOUSE AS T2
 INNER JOIN _mergesource T1 ON T1.O_ID = T2.O_ID
                                        AND           T1.O_D_ID = T2.O_D_ID
                                        AND           T1.O_W_ID = T2.O_W_ID


INSERT INTO ORDERS_DATAWAREHOUSE( O_ID, O_D_ID, O_W_ID, O_C_ID, O_ENTRY_D,
O_CARRIER_ID, O_OL_CNT, O_ALL_LOCAL)
SELECT O_ID, O_D_ID, O_W_ID, O_C_ID, O_ENTRY_D, O_CARRIER_ID, O_OL_CNT,
O_ALL_LOCAL FROM  _mergesource WHERE NOT EXISTS( SELECT O_ID, O_D_ID, O_W_ID FROM
ORDERS_DATAWAREHOUSE where O_ID = _mergesource.O_ID AND O_W_ID =
_mergesource.O_W_ID AND O_D_ID = _mergesource.O_D_ID)
```

Join results

Source	SumSq	d.f.	MeanSq	F	Prob>F
size	102,023,341.50	2	51,011,670.75	5,092.24	0.0000
indexing	20,111,704.17	1	20,111,704.17	2,007.65	0.0000
distribution	22,819,290.20	1	22,819,290.20	2,277.94	0.0000
size*indexing	16,152,082.12	2	8,076,041.06	806.19	0.0000
size*distribution	13,451,504.51	2	6,725,752.25	671.40	0.0000
indexing*distribution	86,429,607.88	1	86,429,607.88	8,627.84	0.0000
size*indexing*distribution	93,743,725.06	2	46,871,862.53	4,678.99	0.0000
Error	23,921,850.16	2388	10,017.53		
Total	378,653,105.59	2399			

Subquery - Join Average Difference of Queries(ms)

	Small		Small Total
Distribution	Extensive	Minimal	
NonUniform	10.7(39.1)	-1.9(60.6)	4.4(51.4)
Uniform	-41.9(40.7)	7.5(63.0)	-17.2(58.5)
Grand Total	-15.6(47.8)	2.8(62.0)	-6.4(56.1)

	Medium		Medium Total
Distribution	Extensive	Minimal	
NonUniform	356.6(105.5)	-2.8(86.4)	176.9(204.0)
Uniform	4.7(29.9)	-4.1(104.9)	0.3(77.3)
Grand Total	180.7(192.3)	-3.5(96.1)	88.6(177.7)

	Large		Large Total
Distribution	Extensive	Minimal	
NonUniform	1322.0(147.2)	6.2(144.6)	664.1(673.9)
Uniform	2.8(61.4)	551.6(178.7)	277.2(305.2)
Grand Total	662.4(669.2)	278.9(317.5)	470.7(557.7)

Execution time differences as a percentage of execution time.

	Small		Small Total
Distribution	Extensive	Minimal	
NonUniform	2.72%	-0.12%	2.00%
Uniform	-139.67%	0.61%	-17.2(58.5)
Grand Total	-68.48%	0.24%	-6.4(56.1)

	Medium		Medium Total
Distribution	Extensive	Minimal	
NonUniform	10.85%	-0.02%	5.41%
Uniform	0.87%	-0.04%	0.41%
Grand Total	5.86%	-0.03%	2.91%

	Large		Large Total
Distribution	Extensive	Minimal	
NonUniform	8.25%	0.01%	4.13%
Uniform	0.12%	0.98%	0.55%
Grand Total	4.18%	0.50%	2.34%

Date Results

Source	SumSq	d.f.	Mean Sq	F	Prob>F
size	2,763,327,183.50	2	1,381,663,591.70	43,671.94	0.0000
indexing	2,745,230,785.50	1	2,745,230,785.50	86,771.88	0.0000
distribution	2,221,043.80	1	2,221,043.80	70.20	0.0000
size*indexing	2,644,270,518.20	2	1,322,135,259.10	41,790.35	0.0000
size*distribution	14,821,823.40	2	7,410,911.70	234.25	0.0000
indexing*distribution	662,243.30	1	662,243.30	20.93	0.0000
size*indexing*distribution	11,584,571.20	2	5,792,285.60	183.08	0.0000
Error	113,514,748.50	3588	31,637.30		
Total	8,295,632,917.40	3599			

DatePart-DateBetween Average Difference of Queries(ms)

	Small		Small Total
Uniform	Extensive	Minimal	
NonUniform	141.5(98.6)	-11.8(27.6)	64.9(105.4)
Uniform	93.1(57.1)	3.2(15.3)	48.2(61.4)
Grand Total	117.3(84.1)	-4.3(23.6)	56.6(86.7)

	Medium		Medium Total
Uniform	Extensive	Minimal	
NonUniform	1073.0(150.6)	-0.8(67.2)	536.1(549.4)
Uniform	936.9(113.0)	8.42(41.9)	472.6(472.0)
Grand Total	1004.9(149.5)	3.8(56.2)	504.4(513.2)

	Large		Large Total
Uniform	Extensive	Minimal	
NonUniform	3949.0(492.8)	18.0(117.9)	1983.5(1997.9)
Uniform	4363.9(214.9)	61.3(140.6)	2212.6(2159.0)
Grand Total	4156.5(433.1)	39.7(131.5)	2098.1(2083.1)

Execution time differences as a percentage of execution time.

	Small		Small Total
Uniform	Extensive	Minimal	
NonUniform	59.70%	-3.46%	31.65%
Uniform	50.32%	1.11%	25.72%
Grand Total	55.01%	2.35%	28.68%

	Medium		Medium Total
Uniform	Extensive	Minimal	
NonUniform	40.74%	-0.03%	20.35%
Uniform	25.32%	0.28%	12.80%
Grand Total	33.03%	0.12%	16.58%

	Large		Large Total
Uniform	Extensive	Minimal	
NonUniform	21.53%	0.12%	10.83%
Uniform	23.18%	0.42%	11.80%
Grand Total	22.35%	0.27%	11.31%

Merge Results

Source	SumSq	df	MeanSq	F	Prob>F
size	13,203,233.63	2	6,601,616.81	23.26	0.0000
indexing	1,693,851.59	1	1,693,851.59	5.97	0.0146
distribution	5,807,881.16	1	5,807,881.16	20.47	0.0000
size*indexing	9,957,083.48	2	4,978,541.74	17.54	0.0000
size*distribution	15,223,148.37	2	7,611,574.19	26.82	0.0000
indexing*distribution	1,213,682.00	1	1,213,682.00	4.28	0.0387
size*indexing*distribution	4,516,508.04	2	2,258,254.02	7.96	0.0004
Error	997,704,909.50	3516	283,761.35		
Total	1,049,320,297.78	3527			

Upsert - Merge Average Difference of Queries(ms)

	Small		Small Total
Distribution	Extensive	Minimal	
NonUniform	170.6(46.2)	187.3(52.5)	179.0(50.1)
Uniform	179.5(26.6)	71.4(34.2)	125.4(62.2)
Grand Total	175.1(38.0)	129.3(72.9)	152.2(62.5)

	Medium		Medium Total
Distribution	Extensive	Minimal	
NonUniform	-18.6(77.8)	-34.2(1388.6)	-26.4(983.4)
Uniform	-57.3(760.4)	132.8(696.0)	37.7(735.1)
Grand Total	-38.0(540.8)	49.3(1101.5)	5.7(868.8)

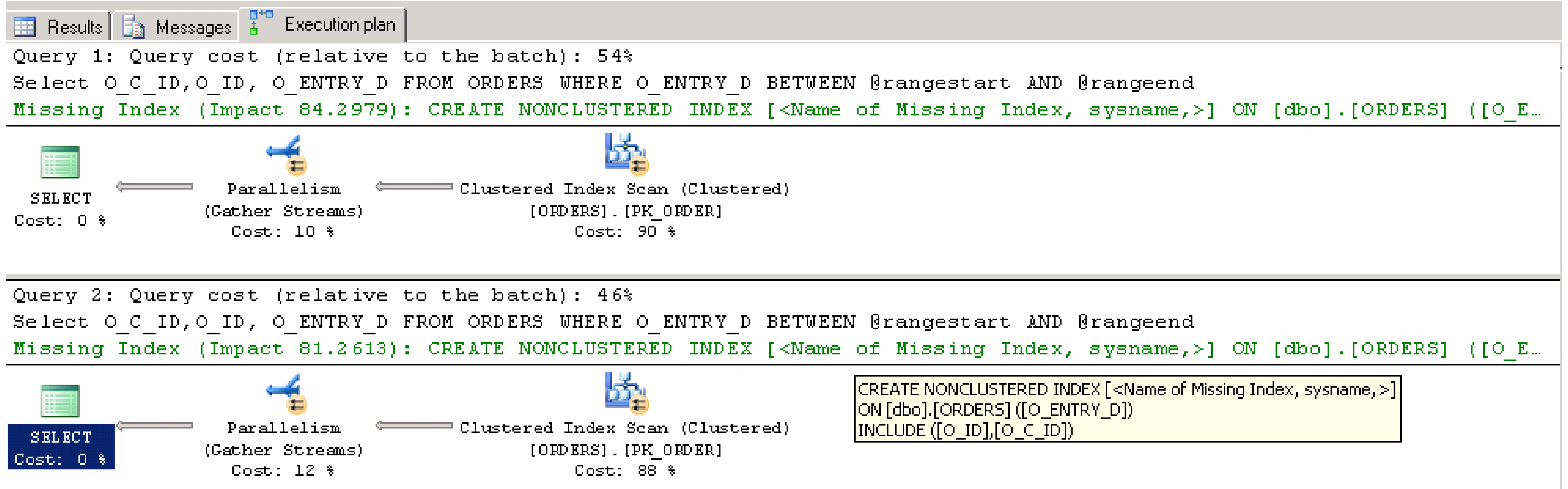
	Large		Large Total
Distribution	Extensive	Minimal	
NonUniform	355.0(381.3)	111.3(251.6)	233.1(345.2)
Uniform	30.1(389.0)	-72.0(173.1)	-20.9(305.4)
Grand Total	192.6(418.0)	19.6(234.6)	106.1(349.8)

Execution time differences as a percentage of execution time.

	Small		Small Total
Distribution	Extensive	Minimal	
NonUniform	62.95%	67.37%	65.16%
Uniform	63.88%	49.58%	56.73%
Grand Total	63.42%	58.48%	60.95%

	Medium		Medium Total
Distribution	Extensive	Minimal	
NonUniform	-2.07%	-5.36%	-3.71%
Uniform	-6.08%	18.32%	6.12%
Grand Total	-4.07%	6.48%	1.20%

	Large		Large Total
Distribution	Extensive	Minimal	
NonUniform	8.47%	4.71%	6.59%
Uniform	0.73%	-2.85%	-1.06%
Grand Total	4.60%	0.93%	2.77%


Results
Messages
Execution plan
Query 1: Query cost (relative to the batch): 54%
Select O_C_ID,O_ID, O_ENTRY_D FROM ORDERS WHERE O_ENTRY_D BETWEEN @rangestart AND @rangeend
Missing Index (Impact 84.2979): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[ORDERS] ([O_E…
SELECT
Cost: 0 %
Parallelism
(Gather Streams)
Cost: 10 %
Clustered Index Scan (Clustered)
[ORDERS].[PK_ORDER]
Cost: 90 %
Query 2: Query cost (relative to the batch): 46%
Select O_C_ID,O_ID, O_ENTRY_D FROM ORDERS WHERE O_ENTRY_D BETWEEN @rangestart AND @rangeend
Missing Index (Impact 81.2613): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].[ORDERS] ([O_E…
SELECT
Cost: 0 %
Parallelism
(Gather Streams)
Cost: 12 %
Clustered Index Scan (Clustered)
[ORDERS].[PK_ORDER]
Cost: 88 %
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [dbo].[ORDERS] ([O_ENTRY_D])
INCLUDE ([O_ID],[O_C_ID])

```
USE [master]
GO

/****** Object:  Database [TPC]    Script Date: 04/22/2009 23:25:20 ******/
CREATE DATABASE [TPC] ON  PRIMARY
( NAME = N'TPC', FILENAME = N'E:\SQLDATA\TPC.mdf' , SIZE = 14720KB , MAXSIZE = UNLIMITED, FILEGROWTH = 15%)
 LOG ON
( NAME = N'TPC_log', FILENAME = N'D:\SQLLOGFILES\TPC_1.ldf' , SIZE = 1280KB , MAXSIZE = 2048GB , FILEGROWTH =
15%)
GO

ALTER DATABASE [TPC] SET COMPATIBILITY_LEVEL = 100
GO

IF (1 = FULLTEXTSERVICEPROPERTY('IsFullTextInstalled'))
begin
EXEC [TPC].[dbo].[sp_fulltext_database] @action = 'enable'
end
GO

ALTER DATABASE [TPC] SET ANSI_NULL_DEFAULT OFF
GO

ALTER DATABASE [TPC] SET ANSI_NULLS OFF
GO

ALTER DATABASE [TPC] SET ANSI_PADDING OFF
GO

ALTER DATABASE [TPC] SET ANSI_WARNINGS OFF
GO

ALTER DATABASE [TPC] SET ARITHABORT OFF
GO

ALTER DATABASE [TPC] SET AUTO_CLOSE OFF
GO

ALTER DATABASE [TPC] SET AUTO_CREATE_STATISTICS ON
GO

ALTER DATABASE [TPC] SET AUTO_SHRINK OFF
GO

ALTER DATABASE [TPC] SET AUTO_UPDATE_STATISTICS ON
GO

ALTER DATABASE [TPC] SET CURSOR_CLOSE_ON_COMMIT OFF
GO

ALTER DATABASE [TPC] SET CURSOR_DEFAULT  GLOBAL
GO

ALTER DATABASE [TPC] SET CONCAT_NULL_YIELDS_NULL OFF
GO

ALTER DATABASE [TPC] SET NUMERIC_ROUNDABORT OFF
GO

ALTER DATABASE [TPC] SET QUOTED_IDENTIFIER OFF
GO

ALTER DATABASE [TPC] SET RECURSIVE_TRIGGERS OFF
GO

ALTER DATABASE [TPC] SET  DISABLE_BROKER
GO

ALTER DATABASE [TPC] SET AUTO_UPDATE_STATISTICS_ASYNC OFF
GO
```

```
ALTER DATABASE [TPC] SET DATE_CORRELATION_OPTIMIZATION OFF
GO

ALTER DATABASE [TPC] SET TRUSTWORTHY OFF
GO

ALTER DATABASE [TPC] SET ALLOW_SNAPSHOT_ISOLATION OFF
GO

ALTER DATABASE [TPC] SET PARAMETERIZATION SIMPLE
GO

ALTER DATABASE [TPC] SET READ_COMMITTED_SNAPSHOT OFF
GO

ALTER DATABASE [TPC] SET HONOR_BROKER_PRIORITY OFF
GO

ALTER DATABASE [TPC] SET  READ_WRITE
GO

ALTER DATABASE [TPC] SET RECOVERY FULL
GO

ALTER DATABASE [TPC] SET  MULTI_USER
GO

ALTER DATABASE [TPC] SET PAGE_VERIFY CHECKSUM
GO

ALTER DATABASE [TPC] SET DB_CHAINING OFF
GO
```

```
exec uspPopulateTPCTables
            @randomSeed,  --Seed for random data generation
            @numWarehouses,--Number of Warehouses
            @numDistricts, --Number of Districts per Warehouses
            @numCustomers, --Number of Customers per District
            @numOrders,    --Number of Orders per Customer
            @minOrderDate, --Lower bound of O_ENTRY_D in Orders
            @maxOrderDate, --Upper bound of O_ENTRY_D in Orders
            @yearsOfSamples, --Number of years to sample for Date and Merge Experiments
            @rangeOfDates,   --Number of months of Order data to include in the Merge Experiment
            @overlappingDate, --Number of months of overlapping data in the source and target tables
                              --in the merge experiment
            @itemSamples,    --Number of samples for the join experiment
            @indexingScheme, --'m' for minimal or 'e' for extensive indexing
            @distribution    --'u' for uniform or 'n' for nonuniform

USE TPC_Small_M_U --Small Databases contain 100,000 Orders and approximately 1,000,000 OrderLines
exec uspPopulateTPCTables 400,  1, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'm', 'u'
USE TPC_Small_E_U
exec uspPopulateTPCTables 400,  1, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'e', 'u'
USE TPC_Small_M_N
exec uspPopulateTPCTables 400,  1, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'm', 'n'
USE TPC_Small_E_N
exec uspPopulateTPCTables 400,  1, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'e', 'n'
USE TPC_Medium_M_U --Medium Databases contain 1,000,000 Orders and approximately 10,000,000 OrderLines
exec uspPopulateTPCTables 600, 10, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'm', 'u'
USE TPC_Medium_E_U
exec uspPopulateTPCTables 600, 10, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'e', 'u'
USE TPC_Medium_M_N
exec uspPopulateTPCTables 600, 10, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'm', 'n'
USE TPC_Medium_E_N
exec uspPopulateTPCTables 600, 10, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'e', 'n'
USE TPC_Large_M_U --Large Databases contain 500,000 Orders and approximately 50,000,000 OrderLines
exec uspPopulateTPCTables 500, 50, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'm', 'u'
USE TPC_Large_E_U
exec uspPopulateTPCTables 500, 50, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'e', 'u'
USE TPC_Large_M_N
exec uspPopulateTPCTables 500, 50, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'm', 'n'
USE TPC_Large_E_N
exec uspPopulateTPCTables 500, 50, 10, 100, 100, '01/01/1969', '01/01/2009', 25, 6, 3, 200, 'e', 'n'
```

```
USE [TPC]
GO
/****** Object:  StoredProcedure [dbo].[uspRandomPrice]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspRandomPrice]
/***********************************
Creates a uniform random number for use in a
variety of other stored procedures
************************************/
@min numeric(6,2),
@max numeric(6,2),
@RandomPrice numeric(6,2) OUTPUT
AS
	declare @range numeric(6,2)
	SET @range = (@max - @min)
	declare @rand numeric(6,2)
	set @rand = Rand()
	SET @RandomPrice =  ((@rand * @range) + @min)
GO
/****** Object:  StoredProcedure [dbo].[uspRandomInt]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspRandomInt]
@min int,
@max int,
@RandomInt int OUTPUT
AS
/***********************************
Generate a uniform random integer
between the @min and @max inputs
**********************************/
	declare @range int
	SET @range = (@max - @min) + 1
	declare @randFloat float
	set @randFloat = Rand()
	SET @RandomInt =  ((@randFloat * @range) + @min)
GO
/****** Object:  StoredProcedure [dbo].[uspNURand]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROC [dbo].[uspNURand]
@A int,
@x int,
@y int,
@C int,
@NURand int OUTPUT
	AS
/**************************
Generates a Nonuniform random number
as described in TPC-C 2.1.6


**************************/
		DECLARE @randAX float
		DECLARE @randXY float
		SELECT  @randAX = Rand()
		SELECT  @randXY = Rand()
		SET  @randXY = (@randXY * (@y - @x)) + 1
		SET  @randAX = (@randAX * (@x - 0)) + 1
		SET @NURand = ((CAST(@randXY as int) | ((CAST(@randAX as int)+ @C))) % (@y-@x+1) + @x)
GO
/****** Object:  StoredProcedure [dbo].[uspMakeZip]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
```

```
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROC [dbo].[uspMakeZip]
@suffix varchar(9),
@zip varchar(9) OUTPUT
AS
/*****************************
Generates a zip code according to TPC-C
*****************************/
DECLARE @rand float
DECLARE @randInt int
SELECT @rand = Rand()
SET @randInt = @rand * 10000
while @randInt < 1000
BEGIN
set @randInt = @randInt * 10
END
set @zip =  CONVERT(varchar(4), @randInt) + @suffix
GO
/****** Object:  StoredProcedure [dbo].[uspMakePercentage]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspMakePercentage]
@threshold float,
@percentage numeric(4,4) OUTPUT
AS
/*************************************
Return a percentage less than @threshold
Used for random data generation in TPC-C

*************************************/
                DECLARE @rand float
                SELECT @rand = Rand()
                while @rand > @threshold
                SET @rand = @rand/2
                SET @percentage = @rand
GO
/****** Object:  StoredProcedure [dbo].[uspMakeCreditRating]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspMakeCreditRating]
@creditrating varchar(2) OUTPUT
AS
/*******************************************
Create a credit rating for a customer
according to TPC-C.  10% should have a BC rating


*******************************************/
                DECLARE @rand float
                SET @creditrating = 'GC'
                SELECT @rand = Rand()
                if @rand > 0.9
                SET @creditrating = 'BC'
GO
/****** Object:  View [dbo].[view_dmv]    Script Date: 04/22/2009 23:12:30 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE VIEW [dbo].[view_dmv]
AS
SELECT     p.name, ps.database_id, ps.object_id, ps.sql_handle, ps.plan_handle, ps.cached_time,
ps.last_execution_time, ps.execution_count,
```

```
                    ps.total_worker_time, ps.last_worker_time, ps.min_worker_time, ps.max_worker_time,
ps.total_physical_reads, ps.last_physical_reads,
                    ps.min_physical_reads, ps.max_physical_reads, ps.total_logical_writes, ps.last_logical_writes,
ps.min_logical_writes, ps.max_logical_writes,
                    ps.total_logical_reads, ps.last_logical_reads, ps.min_logical_reads, ps.max_logical_reads,
ps.total_elapsed_time, ps.last_elapsed_time,
                    ps.min_elapsed_time, ps.max_elapsed_time
FROM        sys.procedures AS p INNER JOIN
                    sys.dm_exec_procedure_stats AS ps ON ps.object_id = p.object_id
WHERE     (ps.database_id =
                        (SELECT     dbid
                          FROM          master.sys.sysprocesses
                          WHERE      (spid = @@SPID)))
GO
EXEC sys.sp_addextendedproperty @name=N'MS_DiagramPane1', @value=N'[0E232FF0-B466-11cf-A24F-
00AA00A3EFFF, 1.00]
Begin DesignProperties =
   Begin PaneConfigurations =
      Begin PaneConfiguration = 0
         NumPanes = 4
         Configuration = "(H (1[40] 4[20] 2[20] 3) )"
      End
      Begin PaneConfiguration = 1
         NumPanes = 3
         Configuration = "(H (1 [50] 4 [25] 3))"
      End
      Begin PaneConfiguration = 2
         NumPanes = 3
         Configuration = "(H (1 [50] 2 [25] 3))"
      End
      Begin PaneConfiguration = 3
         NumPanes = 3
         Configuration = "(H (4 [30] 2 [40] 3))"
      End
      Begin PaneConfiguration = 4
         NumPanes = 2
         Configuration = "(H (1 [56] 3))"
      End
      Begin PaneConfiguration = 5
         NumPanes = 2
         Configuration = "(H (2 [66] 3))"
      End
      Begin PaneConfiguration = 6
         NumPanes = 2
         Configuration = "(H (4 [50] 3))"
      End
      Begin PaneConfiguration = 7
         NumPanes = 1
         Configuration = "(V (3))"
      End
      Begin PaneConfiguration = 8
         NumPanes = 3
         Configuration = "(H (1[56] 4[18] 2) )"
      End
      Begin PaneConfiguration = 9
         NumPanes = 2
         Configuration = "(H (1 [75] 4))"
      End
      Begin PaneConfiguration = 10
         NumPanes = 2
         Configuration = "(H (1[66] 2) )"
      End
      Begin PaneConfiguration = 11
         NumPanes = 2
         Configuration = "(H (4 [60] 2))"
      End
      Begin PaneConfiguration = 12
         NumPanes = 1
         Configuration = "(H (1) )"
      End
```

```
      Begin PaneConfiguration = 13
         NumPanes = 1
         Configuration = "(V (4))"
      End
      Begin PaneConfiguration = 14
         NumPanes = 1
         Configuration = "(V (2))"
      End
      ActivePaneConfig = 0
   End
   Begin DiagramPane =
      Begin Origin =
         Top = 0
         Left = 0
      End
      Begin Tables =
         Begin Table = "p"
            Begin Extent =
               Top = 6
               Left = 38
               Bottom = 114
               Right = 233
            End
            DisplayFlags = 280
            TopColumn = 0
         End
         Begin Table = "ps"
            Begin Extent =
               Top = 6
               Left = 271
               Bottom = 114
               Right = 450
            End
            DisplayFlags = 280
            TopColumn = 0
         End
      End
   End
   Begin SQLPane =
   End
   Begin DataPane =
      Begin ParameterDefaults = ""
      End
      Begin ColumnWidths = 9
         Width = 284
         Width = 1500
         Width = 1500
         Width = 1500
         Width = 1500
         Width = 1500
         Width = 1500
         Width = 1500
         Width = 1500
      End
   End
   Begin CriteriaPane =
      Begin ColumnWidths = 11
         Column = 1440
         Alias = 900
         Table = 1170
         Output = 720
         Append = 1400
         NewValue = 1170
         SortType = 1350
         SortOrder = 1410
         GroupBy = 1350
         Filter = 1350
         Or = 1350
         Or = 1350
         Or = 1350
```

```
    End
  End
End
' , @level0type=N'SCHEMA',@level0name=N'dbo', @level1type=N'VIEW',@level1name=N'view_dmv'
GO
EXEC sys.sp_addextendedproperty @name=N'MS_DiagramPaneCount', @value=1 ,
@level0type=N'SCHEMA',@level0name=N'dbo', @level1type=N'VIEW',@level1name=N'view_dmv'
GO
/****** Object:  Table [dbo].[ITEM]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[ITEM](
	[I_ID] [int] IDENTITY(1,1) NOT NULL,
	[I_IM_ID] [int] NULL,
	[I_NAME] [char](24) NULL,
	[I_PRICE] [numeric](5, 2) NOT NULL,
	[I_DATA] [char](50) NULL,
 CONSTRAINT [PK_ITEM] PRIMARY KEY CLUSTERED
(
	[I_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[_mergequarters]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[_mergequarters](
	[ID] [int] IDENTITY(1,1) NOT NULL,
	[StartDate] [date] NOT NULL,
	[EndDate] [date] NOT NULL,
	[StartDate2] [date] NULL,
	[EndDate2] [date] NULL,
 CONSTRAINT [PK__mergequarters] PRIMARY KEY CLUSTERED
(
	[ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[_itemsample]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[_itemsample](
	[ID] [int] NOT NULL,
	[ItemID] [int] NOT NULL,
 CONSTRAINT [PK__itemsample] PRIMARY KEY CLUSTERED
(
	[ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[_dateranges]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[_dateranges](
	[ID] [int] NOT NULL,
```

```
            [StartTime] [date] NOT NULL,
            [EndTime] [date] NOT NULL,
 CONSTRAINT [PK__dateranges] PRIMARY KEY CLUSTERED
(
            [ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  StoredProcedure [dbo].[_addIndexes]    Script Date: 04/22/2009 23:12:26 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROC [dbo].[_addIndexes]
AS


SET QUOTED_IDENTIFIER ON
SET ARITHABORT ON
SET NUMERIC_ROUNDABORT OFF
SET CONCAT_NULL_YIELDS_NULL ON
SET ANSI_NULLS ON
SET ANSI_PADDING ON
SET ANSI_WARNINGS ON

CREATE NONCLUSTERED INDEX [IX_ORDERS] ON [dbo].[ORDERS]
(
            [O_ENTRY_D] ASC
) WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON,
FILLFACTOR = 80) ON [PRIMARY]



CREATE NONCLUSTERED INDEX [IX_ORDERS_1] ON [dbo].[ORDERS]
(
                        [O_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON,
FILLFACTOR = 80) ON [PRIMARY]


CREATE NONCLUSTERED INDEX [IX_ORDERLINE] ON [dbo].[ORDERLINE]
(
            [OL_O_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON,
FILLFACTOR = 80) ON [PRIMARY]


CREATE NONCLUSTERED INDEX [IX_ORDERLINE_ItemID] ON [dbo].[ORDERLINE]
(
            [OL_I_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON,
FILLFACTOR = 80) ON [PRIMARY]


CREATE NONCLUSTERED INDEX [IX_ORDERS_DW] ON [dbo].[ORDERS_DATAWAREHOUSE]
(
            [O_ENTRY_D] ASC
) WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON,
FILLFACTOR = 80) ON [PRIMARY]



CREATE NONCLUSTERED INDEX [IX_ORDERS_DW1] ON [dbo].[ORDERS_DATAWAREHOUSE]
(
```

```
				[O_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON,
FILLFACTOR = 80) ON [PRIMARY]


CREATE NONCLUSTERED INDEX [IX_mergesource] ON [dbo].[_mergesource]
(
	[O_ENTRY_D] ASC
) WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON,
FILLFACTOR = 80) ON [PRIMARY]



CREATE NONCLUSTERED INDEX [IX_mergesource_1] ON [dbo].[_mergesource]
(
				[O_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY
= OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON,
FILLFACTOR = 80) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[_Syllables]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[_Syllables](
	[ind] [int] NULL,
	[syl] [varchar](5) NULL
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[_Results_MergeExperiment]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[_Results_MergeExperiment](
	[runID] [int] NOT NULL,
	[StartTime] [date] NOT NULL,
	[EndTime] [date] NOT NULL,
	[RunType] [nvarchar](50) NOT NULL,
	[StopWatchTime] [datetime] NOT NULL,
	[dmvTime] [int] NOT NULL,
	[PhysicalReads] [int] NOT NULL,
	[LogicalReads] [int] NOT NULL,
	[LogicalWrites] [int] NOT NULL,
	[TargetInserts] [int] NULL,
	[TargetDeletes] [int] NULL,
	[TargetUpdates] [int] NULL
) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[_Results_JoinExperiment]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[_Results_JoinExperiment](
	[runID] [int] NOT NULL,
	[RunType] [nvarchar](50) NOT NULL,
	[ItemId] [int] NOT NULL,
	[NumRecords] [int] NOT NULL,
	[StopWatchTime] [datetime] NOT NULL,
	[dmvTime] [int] NOT NULL,
	[PhysicalReads] [int] NOT NULL,
	[LogicalReads] [int] NOT NULL,
```

```
	[LogicalWrites] [int] NOT NULL
) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[_Results_DateExperiment]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[_Results_DateExperiment](
	[runID] [int] NOT NULL,
	[StartTime] [datetime] NOT NULL,
	[EndTime] [datetime] NOT NULL,
	[RunType] [nvarchar](50) NOT NULL,
	[NumRecords] [int] NOT NULL,
	[StopWatchTime] [datetime] NOT NULL,
	[dmvTime] [int] NOT NULL,
	[PhysicalReads] [int] NOT NULL,
	[LogicalReads] [int] NOT NULL,
	[LogicalWrites] [int] NOT NULL
) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[DISTRICT]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[DISTRICT](
	[D_ID] [int] NOT NULL,
	[D_W_ID] [int] NOT NULL,
	[D_NAME] [char](10) NOT NULL,
	[D_STREET_1] [char](20) NOT NULL,
	[D_STREET_2] [char](20) NULL,
	[D_CITY] [char](20) NOT NULL,
	[D_STATE] [varchar](2) NOT NULL,
	[D_ZIP] [varchar](9) NOT NULL,
	[D_TAX] [numeric](4, 4) NOT NULL,
	[D_YTD] [numeric](12, 2) NOT NULL,
	[D_NEXT_O_ID] [int] NOT NULL,
 CONSTRAINT [PK_DISTRICT_1] PRIMARY KEY CLUSTERED 
(
	[D_ID] ASC,
	[D_W_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF, 
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[STOCK]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[STOCK](
	[S_I_ID] [int] NOT NULL,
	[S_W_ID] [int] NOT NULL,
	[S_QUANTITY] [numeric](4, 0) NOT NULL,
	[S_DIST_01] [varchar](24) NOT NULL,
	[S_DIST_02] [varchar](24) NOT NULL,
	[S_DIST_03] [varchar](24) NOT NULL,
	[S_DIST_04] [varchar](24) NOT NULL,
	[S_DIST_05] [varchar](24) NOT NULL,
	[S_DIST_06] [varchar](24) NOT NULL,
	[S_DIST_07] [varchar](24) NOT NULL,
	[S_DIST_08] [varchar](24) NOT NULL,
	[S_DIST_09] [varchar](24) NOT NULL,
```

```
	[S_DIST_10] [varchar](24) NOT NULL,
	[S_YTD] [numeric](8, 0) NOT NULL,
	[S_ORDER_CNT] [numeric](4, 0) NOT NULL,
	[S_REMOTE_CNT] [numeric](4, 0) NOT NULL,
	[S_DATA] [char](50) NULL,
 CONSTRAINT [PK_STOCK] PRIMARY KEY CLUSTERED
(
	[S_I_ID] ASC,
	[S_W_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
/****** Object:  StoredProcedure [dbo].[LName]    Script Date: 04/22/2009 23:12:26 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[LName]
/*********************************
Procedure to generate a last name
as instructed in TPCC 2.1.6
*********************************/
@lname varchar(18) OUTPUT
AS
	DECLARE @nurand int
	DECLARE @mod int
	DECLARE @hundreds int
	DECLARE @tens int
	DECLARE @ones int
	SET @nurand =  dbo.NURand(255,0,999,166)

	SET @hundreds =   @nurand /  100
	SET @mod = @nurand % 100
	SET @tens = @mod / 10
	SET @ones = @mod % 10

	DECLARE @first varchar(6)
	DECLARE @middle varchar(6)
	DECLARE @last varchar(6)

	SELECT @first = syl FROM dbo._Syllables WHERE ind = @hundreds
	SELECT @middle = syl FROM dbo._Syllables WHERE ind = @tens
	SELECT @last = syl FROM dbo._Syllables WHERE ind = @ones
	SET @lname = @first+@middle+@last
GO
/****** Object:  StoredProcedure [dbo].[uspRandomString]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspRandomString]
@minlength int,
@maxlength int,
@asciimin int,
@asciimax int,
@Str varchar(500) OUTPUT
AS
/**********************************************
Generates a random string of characters using ASCII
values between @asciimin and @asciimax, and a length between
@minlength and @maxlength
**********************************************/
		declare @asciival int
		declare @charToAdd char
		Set @Str = ''
		declare @counter int
		declare @numOfChars int
```

```
			--obtains the length of the string
			exec dbo.uspRandomInt @minlength, @maxlength, @RandomInt = @numOfChars OUTPUT

			--fill the string with random characters
			SET @counter = 0
			while @counter < @numOfChars
			begin
				exec dbo.uspRandomInt @asciimin, @asciimax, @RandomInt = @asciival OUTPUT
				set @charToAdd = char(@asciival)
				set @Str = @Str + @charToAdd
				SET @counter = @counter + 1
			end
GO
/****** Object:  StoredProcedure [dbo].[uspCreateQuartersDataSet]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROC [dbo].[uspCreateQuartersDataSet]
@span int,
@range int
AS
/*****************************************
Uses _dateranges table (containing first and last day of each month)
to create a table where each record contains 2 sets of date ranges.  @span is
the number of months in the range.  @range is the number of months in the intersection
of the two date ranges.  For example

StartDate EndDate      StartDate2         EndDate2
2008-02-01          2008-08-31          2008-05-01          2008-11-30

Was originally used to generate date ranges of yearly quarters, but parameters
were added to allow for variation.
*****************************************/
delete from _mergequarters
declare @start int
--Get the number of records in the _dateranges table.  This will be the ID of the starting point
SELECT @start = COUNT(*) FROM _dateranges
--Subtract @span from @start to get the max value of the range
declare @end int = @start - @span
declare @startdate date
declare @enddate date
declare @startdate2 date
declare @enddate2 date
while @end >0
BEGIN
			SELECT @startdate = StartTime FROM _dateranges where ID = @start
			SELECT @enddate = EndTime FROM _dateranges where ID = @end
			--Subtract the number of range months to get the other date range
			SELECT @startdate2 = StartTime FROM _dateranges where ID = @start - @range
			SELECT @enddate2 = EndTime FROM _dateranges where ID = @end - @range
			INSERT INTO dbo._mergequarters (StartDate, EndDate, StartDate2, EndDate2) VALUES
(@startdate , @enddate, @startdate2, @enddate2)
		SET @start = @start - 1
		SET @end = @end - 1
END
GO
/****** Object:  StoredProcedure [dbo].[uspCreateDateDataSet]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspCreateDateDataSet]
@datestart datetime,
@years int
AS
/*****************************************
Accepts a start day and number of years (n) as parameters.
Inserts a row into _dateranges with the first and last day
of each month beginning with the parameter month and working
```

```
backwards for n years.

*******************************************/
SET NOCOUNT ON
delete from _dateranges
declare @date datetime
set @date = @datestart
--Total number of months to insert
declare @months int = @years*12
declare  @dateend datetime

declare @i int = 0
while @i < 12*@years
BEGIN
set @i = @i + 1

select @date = dateadd(m, datediff(m, 0, dateadd(m, 0, @date)), -1) --end of month

select @dateend = dateadd(m, datediff(m, 0, @date), 0) -- start of month

 INSERT INTO _dateranges (ID,  StartTime, EndTime) VALUES (@i, @dateend, @date)
END
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateWarehouse]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateWarehouse]
        @numWarehouses int
        AS

/****************************************
Add Warehouses to the WAREHOUSE table as described in TPC-C

****************************************/
        DELETE  FROM dbo.WAREHOUSE
        --Min and Max ASCII characters to use in data generation
        declare @charsetmin int = 32
        declare @charsetmax int = 160

        DECLARE @i int
        SET @i = 0
        WHILE @i < @numWarehouses
        BEGIN
                SET @i = @i+1
        declare @name varchar(10)
                exec dbo.uspRandomString 6, 10, @charsetmin, @charsetmax, @Str = @name OUTPUT
        declare @street1 varchar(20)
                exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @street1 OUTPUT
        declare @street2 varchar(20)
                exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @street2 OUTPUT
        declare @city varchar(20)
                exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @city OUTPUT
        declare @state varchar(2)
                exec dbo.uspRandomString 2, 2, @charsetmin, @charsetmax, @Str = @state OUTPUT
        declare @zip varchar(9)
                exec dbo.uspMakeZip '11111', @zip = @zip OUTPUT
        declare @tax numeric(4,4)
                exec dbo.uspMakePercentage 0.2, @percentage = @tax OUTPUT

                INSERT INTO dbo.WAREHOUSE
                        (W_ID, W_NAME, W_STREET_1, W_STREET_2, W_CITY, W_STATE, W_ZIP, W_TAX,
W_YTD)
                        VALUES
                        (@i, @name, @street1, @street2, @city, @state, @zip, @tax,300000)
        END
        RETURN
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateItem]    Script Date: 04/22/2009 23:12:27 ******/
```

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateItem]
/****************************************
Inserts @numItems into dbo.ITEM as described in
TPC-C
*****************************************/
	@numItems int
	AS
	DELETE  FROM dbo.ITEM
	--Declare Min and Max ASCII characters and Min and Max Price values
	declare @charsetmin int = 32
	declare @charsetmax int = 160
	declare @min numeric(5,2) = 1.00
	declare @max numeric(5,2) = 100.00
	declare @randomInt int

	DECLARE @i int
	SET @i = 0
	WHILE @i < @numItems
	BEGIN
		SET @i = @i+1
		declare @name char(24)
		exec dbo.uspRandomString 14, 24, @charsetmin, @charsetmax, @Str = @name OUTPUT
		declare @data char(50)
		exec dbo.uspRandomString 26, 50, @charsetmin, @charsetmax, @Str = @data OUTPUT

		exec dbo.uspRandomInt 1, 10, @RandomInt = @randomInt OUTPUT
		if(		@randomInt = 1)
		BEGIN
			declare @insertPoint int
			declare @length int = len(@data) - 8
			exec dbo.uspRandomInt 0, @length, @RandomInt = @insertPoint OUTPUT
			set @data =		stuff(@data, @insertPoint, 8, 'original')
		END

		declare @price numeric(5,2)
		exec dbo.uspRandomPrice @min, @max, @RandomPrice = @price OUTPUT

		INSERT INTO dbo.ITEM
			( I_IM_ID, I_NAME, I_PRICE, I_DATA)
			VALUES
			( @i, @name, @price, @data)
	END
	RETURN
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateDistrict]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateDistrict]
/***************************************
Add districts to the DISTRICTS table as described in TPC-C

****************************************/
	@numDistrictsPerWarehouse int
	AS
	DELETE  FROM dbo.district
	--Declare upper and lower bounds of ASCII character set to use for data generation
	DECLARE @charsetmin int = 32
	DECLARE @charsetmax int = 160

	DECLARE		@minID int
	DECLARE		@i int

	SET @i=0
	--Insert @numDistrictsPerWarehouse for each Warehouse
```

```
	WHILE @i < @numDistrictsPerWarehouse
	BEGIN
	SET @i = @i+1
	SELECT @minID=MIN(W_ID)
	FROM dbo.WAREHOUSE

	WHILE @minID IS NOT NULL
	BEGIN

	declare @name varchar(10)
		exec dbo.uspRandomString 6, 10, @charsetmin, @charsetmax, @Str = @name OUTPUT
	declare @street1 varchar(20)
		exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @street1 OUTPUT
	declare @street2 varchar(20)
		exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @street2 OUTPUT
	declare @city varchar(20)
		exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @city OUTPUT
	declare @state varchar(2)
		exec dbo.uspRandomString 2, 2, @charsetmin, @charsetmax, @Str = @state OUTPUT
	declare @zip varchar(9)
		exec dbo.uspMakeZip '11111', @zip = @zip OUTPUT
	declare @tax numeric(4,4)
		exec dbo.uspMakePercentage 0.2, @percentage = @tax OUTPUT

			INSERT INTO dbo.DISTRICT
			(D_ID, D_W_ID, D_NAME, D_STREET_1, D_STREET_2, D_CITY, D_STATE, D_ZIP,
D_TAX, D_YTD, D_NEXT_O_ID)
			VALUES
			(@i, @minID,@name, @street1, @street2, @city, @state, @zip,@tax,30000, 1)

		SELECT @minID=MIN(W_ID)
		FROM dbo.WAREHOUSE
		WHERE W_ID > @minID
	END
	END
RETURN
GO
/****** Object:  Table [dbo].[HISTORY]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[HISTORY](
	[H_C_ID] [int] NOT NULL,
	[H_C_D_ID] [int] NOT NULL,
	[H_C_W_ID] [int] NOT NULL,
	[H_D_ID] [int] NOT NULL,
	[H_W_ID] [int] NOT NULL,
	[H_DATE] [datetime] NOT NULL,
	[H_AMOUNT] [numeric](6, 2) NOT NULL,
	[H_DATA] [char](24) NULL
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[CUSTOMER]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[CUSTOMER](
	[C_W_ID] [int] NOT NULL,
	[C_D_ID] [int] NOT NULL,
	[C_ID] [int] NOT NULL,
	[C_FIRST] [char](16) NOT NULL,
	[C_MIDDLE] [varchar](2) NULL,
```

```
	[C_LAST] [char](16) NOT NULL,
	[C_STREET_1] [char](20) NOT NULL,
	[C_STREET_2] [char](20) NULL,
	[C_CITY] [char](20) NOT NULL,
	[C_STATE] [varchar](2) NOT NULL,
	[C_ZIP] [varchar](9) NOT NULL,
	[C_PHONE] [varchar](16) NOT NULL,
	[C_SINCE] [datetime] NOT NULL,
	[C_CREDIT] [varchar](2) NOT NULL,
	[C_CREDIT_LIM] [numeric](12, 2) NOT NULL,
	[C_DISCOUNT] [numeric](4, 4) NOT NULL,
	[C_BALANCE] [numeric](12, 2) NOT NULL,
	[C_YTD_PAYMENT] [numeric](12, 2) NOT NULL,
	[C_PAYMENT_CNT] [numeric](4, 0) NOT NULL,
	[C_DELIVERY_CNT] [numeric](4, 0) NOT NULL,
	[C_DATA] [char](500) NULL,
 CONSTRAINT [PK_CUSTOMER] PRIMARY KEY CLUSTERED
(
	[C_ID] ASC,
	[C_D_ID] ASC,
	[C_W_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[_mergesource]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[_mergesource](
	[O_ID] [int] NOT NULL,
	[O_D_ID] [int] NOT NULL,
	[O_W_ID] [int] NOT NULL,
	[O_C_ID] [int] NOT NULL,
	[O_ENTRY_D] [datetime] NOT NULL,
	[O_CARRIER_ID] [int] NULL,
	[O_OL_CNT] [numeric](2, 0) NOT NULL,
	[O_ALL_LOCAL] [numeric](1, 0) NULL,
 CONSTRAINT [PK_merge] PRIMARY KEY CLUSTERED
(
	[O_ID] ASC,
	[O_D_ID] ASC,
	[O_W_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[ORDERS_DATAWAREHOUSE]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[ORDERS_DATAWAREHOUSE](
	[O_ID] [int] NOT NULL,
	[O_D_ID] [int] NOT NULL,
	[O_W_ID] [int] NOT NULL,
	[O_C_ID] [int] NOT NULL,
	[O_ENTRY_D] [datetime] NOT NULL,
	[O_CARRIER_ID] [int] NULL,
	[O_OL_CNT] [numeric](2, 0) NOT NULL,
	[O_ALL_LOCAL] [numeric](1, 0) NULL,
 CONSTRAINT [PK_ORDERWAREHOUSE] PRIMARY KEY CLUSTERED
(
	[O_ID] ASC,
	[O_D_ID] ASC,
	[O_W_ID] ASC
```

```
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  Table [dbo].[ORDERS]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[ORDERS](
	[O_ID] [int] NOT NULL,
	[O_D_ID] [int] NOT NULL,
	[O_W_ID] [int] NOT NULL,
	[O_C_ID] [int] NOT NULL,
	[O_ENTRY_D] [datetime] NOT NULL,
	[O_CARRIER_ID] [int] NULL,
	[O_OL_CNT] [numeric](2, 0) NOT NULL,
	[O_ALL_LOCAL] [numeric](1, 0) NULL,
 CONSTRAINT [PK_ORDER] PRIMARY KEY CLUSTERED
(
	[O_ID] ASC,
	[O_D_ID] ASC,
	[O_W_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateCustomer]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateCustomer]
@numCustomersPerDistrict int
AS
/**************************************
Add Customers to CUSTOMER table as described in TPC-C


**************************************/

DELETE FROM dbo.CUSTOMER
declare @DID int
declare @WID int
DECLARE @lastname varchar(18)
	declare @i int
	declare CustList cursor for
	select D_ID, D_W_ID from dbo.District ORDER BY D_W_ID
		OPEN CustList
	FETCH NEXT FROM CustList
	INTO @DID, @WID
	--Use cursor to obtain warehouse and district information.
	--As long as there is a district insert Customers
	WHILE @@FETCH_STATUS = 0
	BEGIN
	  set @i = 0

	  declare @charsetmin int = 32
	  declare @charsetmax int = 160
	  declare @first varchar(16)
		exec dbo.uspRandomString 8, 16, @charsetmin, @charsetmax, @Str = @first OUTPUT
	  declare @middle varchar(2)
		set @middle = '0E'
	  declare @last varchar(16)
		exec dbo.LName @lname = @lastname OUTPUT
	  declare @street1 varchar(20)
		exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @street1 OUTPUT
	  declare @street2 varchar(20)
		exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @street2 OUTPUT
	  declare @city varchar(20)
```

```
                exec dbo.uspRandomString 10, 20, @charsetmin, @charsetmax, @Str = @city OUTPUT
        declare @state varchar(2)
                exec dbo.uspRandomString 2, 2, @charsetmin, @charsetmax, @Str = @state OUTPUT
        declare @zip varchar(9)
                exec dbo.uspMakeZip '11111', @zip = @zip OUTPUT
        declare @phone varchar(16)
                set @phone = '1234567890123456'
        declare @cust_since datetime
        declare @credit varchar(2)
          exec dbo.uspMakeCreditRating @creditrating = @credit OUTPUT
        declare @creditlimit numeric(12,2)
        declare @discount numeric(4,4)
                exec dbo.uspMakePercentage 0.5, @percentage = @discount OUTPUT
        declare @balance numeric(12,2)
        declare @ytd_payment numeric(12,2)
        declare @payment_cnt numeric(4,0)
        declare @delivery_cnt numeric(4,0)
        declare @data char(500)
                exec dbo.uspRandomString 300, 500, @charsetmin, @charsetmax, @Str = @data OUTPUT

        --Create @numCustomersPerDistrict for each District
        WHILE @i < @numCustomersPerDistrict
        BEGIN
                                SET @i = @i+1

                                INSERT INTO dbo.CUSTOMER
                                (C_W_ID, C_D_ID, C_ID, C_FIRST, C_MIDDLE, C_LAST, C_STREET_1,
C_STREET_2,
                                C_CITY, C_STATE, C_ZIP, C_PHONE, C_SINCE, C_CREDIT, C_CREDIT_LIM,
C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_DELIVERY_CNT, C_DATA)
                                VALUES
                                (@WID, @DID, @i, @first, @middle, @lastname,
                                @street1, @street2,
                                @city, @state, @zip,
                                @phone, CURRENT_TIMESTAMP, @credit,  50000.00, @discount, -10.0,
10.00, 1, 0, @data)
                END


        FETCH NEXT FROM CustList
          INTO @DID, @WID

        END
        CLOSE CustList
        DEALLOCATE CustList
GO
/****** Object:  StoredProcedure [dbo].[usp_upsertquery]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_upsertquery]
        -- Add the parameters for the stored procedure here
@runtime DATETIME OUTPUT
AS
BEGIN
declare @start_time DATETIME
declare @end_time DATETIME
select @start_time = GETDATE()
        -- SET NOCOUNT ON added to prevent extra result sets from
        -- interfering with SELECT statements.
DBCC DROPCLEANBUFFERS
UPDATE ORDERS_DATAWAREHOUSE
 SET  ORDERS_DATAWAREHOUSE.O_ALL_LOCAL = T1.O_ALL_LOCAL--  You know the column names, not me !
 FROM  ORDERS_DATAWAREHOUSE AS T2
 INNER JOIN _mergesource T1 ON T1.O_ID = T2.O_ID
                                        AND             T1.O_D_ID = T2.O_D_ID
                                        AND             T1.O_W_ID = T2.O_W_ID

 -- WHERE T1.Column IS NOT NULL
```

```
--select @@ROWCOUNT

INSERT INTO ORDERS_DATAWAREHOUSE( O_ID, O_D_ID, O_W_ID, O_C_ID, O_ENTRY_D, O_CARRIER_ID,
O_OL_CNT, O_ALL_LOCAL)
SELECT O_ID, O_D_ID, O_W_ID, O_C_ID, O_ENTRY_D, O_CARRIER_ID, O_OL_CNT, O_ALL_LOCAL FROM
_mergesource WHERE NOT EXISTS( SELECT O_ID, O_D_ID, O_W_ID FROM ORDERS_DATAWAREHOUSE where
O_ID = _mergesource.O_ID AND O_W_ID = _mergesource.O_W_ID AND O_D_ID = _mergesource.O_D_ID)
--select @@ROWCOUNT
SET @end_time = GETDATE()
SET @runtime = @end_time - @start_time
END
GO
/****** Object:  Table [dbo].[ORDERLINE]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[ORDERLINE](
	[OL_O_ID] [int] NOT NULL,
	[OL_D_ID] [int] NOT NULL,
	[OL_W_ID] [int] NOT NULL,
	[OL_NUMBER] [int] NOT NULL,
	[OL_I_ID] [int] NOT NULL,
	[OL_SUPPLY_W_ID] [int] NOT NULL,
	[OL_DELIVERY_D] [datetime] NULL,
	[OL_QUANTITY] [numeric](2, 0) NOT NULL,
	[OL_AMOUNT] [numeric](6, 2) NOT NULL,
	[OL_DIST_INFO] [varchar](24) NULL,
 CONSTRAINT [PK_ORDER-LINE] PRIMARY KEY CLUSTERED
(
	[OL_O_ID] ASC,
	[OL_D_ID] ASC,
	[OL_W_ID] ASC,
	[OL_NUMBER] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[NEW-ORDER]    Script Date: 04/22/2009 23:12:29 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[NEW-ORDER](
	[NO_O_ID] [int] IDENTITY(1,1) NOT NULL,
	[NO_D_ID] [int] NOT NULL,
	[NO_W_ID] [int] NOT NULL,
 CONSTRAINT [PK_NEW-ORDER] PRIMARY KEY CLUSTERED
(
	[NO_O_ID] ASC,
	[NO_D_ID] ASC,
	[NO_W_ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON, FILLFACTOR = 80) ON [PRIMARY]
) ON [PRIMARY]
GO
/****** Object:  StoredProcedure [dbo].[usp_daterangequery]    Script Date: 04/22/2009 23:12:26 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_daterangequery]
/*******************************************
Query to retrieve orders based on a date range
```

```
******************************************/
@rangestart DATETIME,
@rangeend DATETIME,
@runtime DATETIME OUTPUT,
@NumRecords int OUTPUT
as
declare @start_time DATETIME
declare @end_time DATETIME

select @start_time = GETDATE()

Select O_C_ID,O_ID, O_ENTRY_D
                    FROM ORDERS
                    WHERE O_ENTRY_D BETWEEN @rangestart AND @rangeend

set @NumRecords = @@ROWCOUNT
SET @end_time = GETDATE()
SET @runtime = @end_time - @start_time
--Timing mechanism to verify accuracy of runtimes found in dmv
GO
/****** Object:  StoredProcedure [dbo].[usp_datepartquery]    Script Date: 04/22/2009 23:12:26 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_datepartquery]
@datemonth int,
@dateyear int,
@runtime DATETIME OUTPUT,
@NumRecords int OUTPUT
as
declare @start_time DATETIME
declare @end_time DATETIME
--declare @runtime DATETIME
select @start_time = GETDATE()

Select O_C_ID,O_ID, O_ENTRY_D
                    FROM ORDERS
                    WHERE DATEPART(M, O_ENTRY_D) = @datemonth AND
                       DATEPART(year, O_ENTRY_D) = @dateyear
--SELECT 'elapsed time' = DATEDIFF(ms, @start_time, GETDATE())
set @NumRecords = @@ROWCOUNT
SET @end_time = GETDATE()
SET @runtime = @end_time - @start_time
--INSERT INTO dbo._runtimes(runtime) VALUES (@runtime)



--SELECT O_C_ID, O_ID FROM ORDERS WHERE O_ID IN (SELECT OL_O_ID FROM ORDERLINE WHERE OL_I_ID
= @itemid)
GO
/****** Object:  StoredProcedure [dbo].[usp_mergequery]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_mergequery]
/******************************************
Query to merge records from a source table to
a data warehouse


******************************************/
@runtime DATETIME OUTPUT
as

declare @start_time DATETIME
declare @end_time DATETIME
select @start_time = GETDATE()
```

```
		MERGE ORDERS_DATAWAREHOUSE as TARGET
		USING dbo._mergesource AS SOURCE ON
		(TARGET.O_ID = SOURCE.O_ID AND TARGET.O_D_ID = SOURCE.O_D_ID AND TARGET.O_W_ID =
SOURCE.O_W_ID)
			--If the record is not in the data warehouse, insert it
				WHEN NOT MATCHED THEN INSERT VALUES (SOURCE.O_ID, SOURCE.O_D_ID,
SOURCE.O_W_ID, SOURCE.O_C_ID,

SOURCE.O_ENTRY_D, SOURCE.O_CARRIER_ID, SOURCE.O_OL_CNT, SOURCE.O_ALL_LOCAL)
			--If the record is in the data warehouse with ALL_LOCAL variable equal to 9, update it with the source record
			WHEN MATCHED AND TARGET.O_ALL_LOCAL = 9 THEN UPDATE SET TARGET.O_ALL_LOCAL =
SOURCE.O_ALL_LOCAL
	--MERGE statements must end with a ';'
		;

SET @end_time = GETDATE()
--Timing mechanism to verify accuracy of runtimes found in dmv
SET @runtime = @end_time - @start_time
GO
/****** Object:  StoredProcedure [dbo].[usp_joinquery]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_joinquery]
/******************************************
Query to retrieve orders and customer IDs
for orders containing a particular part


******************************************/
@itemid int,
@runtime DATETIME OUTPUT,
@NumRecords int OUTPUT
as
declare @start_time DATETIME
declare @end_time DATETIME
select @start_time = GETDATE()

SELECT DISTINCT O_C_ID, O_ID
		FROM ORDERS  JOIN
				ORDERLINE
				ON O_ID = OL_O_ID
				WHERE OL_I_ID = @itemid

SET @NumRecords = @@ROWCOUNT
SET @end_time = GETDATE()

--Timing mechanism to verify accuracy of runtimes found in dmv
SET @runtime = @end_time - @start_time
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateNUOrderLine]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateNUOrderLine]
/******************************************
Inserts Order Lines into the ORDERLINES table
******************************************/
@numLines int,
@OID int,
@DID int,
@WID int,
@EntryDate datetime,
@A int,
@C int

AS
/******************************************
```

```
Populates the ORDERLINE table

*******************************************/
declare @orderlineID int = 0
declare @itemID int
declare @amount numeric(6,2)
declare @DistInfo varchar(24)
              WHILE @orderlineID < @numLines
              BEGIN

               set @orderlineID = @orderlineID + 1;
                                   exec dbo.uspRandomPrice 1.00, 9999.99, @RandomPrice = @amount OUTPUT
                                   exec dbo.uspRandomString 24, 24, 32, 160, @Str = @DistInfo OUTPUT

                         exec uspNURand @A, 1, 100000, @C, @itemID OUTPUT
                                              INSERT INTO dbo.ORDERLINE(OL_O_ID,
                                                                                  OL_D_ID, OL_W_ID,
OL_NUMBER, OL_I_ID, OL_SUPPLY_W_ID, OL_DELIVERY_D, OL_QUANTITY, OL_AMOUNT, OL_DIST_INFO )
                                              VALUES
                                              (@OID, @DID, @WID, @orderlineid, @itemid, @WID, @EntryDate, 5,
@amount, @DistInfo )

                                   END
GO
/****** Object:  StoredProcedure [dbo].[usp_subquery]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_subquery]
@itemid int,
@runtime DATETIME OUTPUT,
@NumRecords int OUTPUT
as
/*******************************************
Subquery to retrieve Order ID and Customer
for orders containing a particular item


*******************************************/
declare @start_time DATETIME
declare @end_time DATETIME
select @start_time = GETDATE()

Select DISTINCT O_C_ID,O_ID
                     FROM ORDERS
                     WHERE O_ID
                         IN (SELECT OL_O_ID FROM ORDERLINE WHERE OL_I_ID = @itemid )

SET @NumRecords = @@ROWCOUNT
SET @end_time = GETDATE()
--Timing mechanism to verify accuracy of runtimes found in dmv
SET @runtime = @end_time - @start_time
GO
/****** Object:  StoredProcedure [dbo].[usp_runMergeExperiment]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_runMergeExperiment]
/*******************************************
Procedure to run the Merge Experiment


*******************************************/
as
set Statistics TIME ON
set Statistics IO ON
set NOCOUNT ON
```

```
delete from _Results_MergeExperiment
declare @runID int
declare @physical_reads int
declare @physical_writes int
declare @logical_reads int
declare @logical_writes int
declare @elapsed_time int
declare @runtime datetime
declare @RunType nvarchar(50)
--Procedure names retrieve information from the dynamic management views (dmv)
declare @uspname nvarchar(50)= 'usp_mergequery'
declare @uspsubname nvarchar(50)= 'usp_upsertquery'
declare @StartTime date
declare @EndTime date
declare @StartTime2 date
declare @EndTime2 date
declare @TargetInserts int
declare @TargetDeletes int
declare @TargetUpdates int
--Run Experiment on each row in the merge samples table (_mergequarters)
declare DateList cursor for
        select ID, StartDate, EndDate, StartDate2, EndDate2 from dbo._mergequarters ORDER BY ID
                OPEN DateList
        FETCH NEXT FROM DateList
        INTO @runID, @StartTime, @EndTime, @StartTime2, @EndTime2
        WHILE @@FETCH_STATUS = 0
        BEGIN
--ORDERS_DATAWAREHOUSE serves as the target for the merge and upsert
--_mergesource serves as the source for the merge and upsert
DELETE FROM ORDERS_DATAWAREHOUSE
DELETE FROM dbo._mergesource

--Insert rows into the source and data warehouse tables according to information in the sample row
INSERT INTO ORDERS_DATAWAREHOUSE
Select O_ID, O_D_ID, O_W_ID, O_C_ID, O_ENTRY_D, O_CARRIER_ID, O_OL_CNT,
--Change O_ALL_LOCAL column data for some rows so there is something to update from the source table
CASE WHEN O_OL_CNT < 11 THEN 9 ELSE O_ALL_LOCAL END
        FROM ORDERS where O_ENTRY_D BETWEEN @StartTime AND @EndTime


INSERT INTO dbo._mergesource( O_ID, O_D_ID, O_W_ID, O_C_ID, O_ENTRY_D, O_CARRIER_ID, O_OL_CNT,
O_ALL_LOCAL)
(SELECT * FROM ORDERS WHERE O_ENTRY_D BETWEEN @StartTime2 AND @EndTime2)
(((SELECT O_ID, O_D_ID, O_W_ID FROM ORDERS_DATAWAREHOUSE) EXCEPT (SELECT O_ID, O_D_ID, O_W_ID
FROM _mergesource)))  --123   4396
SET @TargetDeletes = @@ROWCOUNT
(SELECT O_ID, O_D_ID, O_W_ID FROM  _mergesource) EXCEPT (SELECT O_ID, O_D_ID, O_W_ID FROM
ORDERS_DATAWAREHOUSE) --234   4179
SET @TargetInserts = @@ROWCOUNT
SELECT @TargetUpdates = COUNT (*) FROM ORDERS_DATAWAREHOUSE WHERE O_ENTRY_D BETWEEN
@StartTime2 and @EndTime2
DBCC DROPCLEANBUFFERS

        --Run the merge query and capture results from dmv
        exec usp_mergequery @runtime = @runtime OUTPUT
        SET @RunType  = 'MERGE'
        SELECT @physical_reads = last_physical_reads from view_dmv where name = @uspname
        SELECT @logical_reads = last_logical_reads from view_dmv where name = @uspname
        SELECT @logical_writes = last_logical_writes from view_dmv where name = @uspname
        SELECT @elapsed_time = last_elapsed_time from view_dmv where name = @uspname
        insert into _Results_MergeExperiment VALUES (@runID, @StartTime, @EndTime, @RunType, @runtime,
@elapsed_time, @physical_reads,  @logical_reads, @logical_writes ,@TargetInserts,@TargetDeletes,@TargetUpdates)

        --ORDERS_DATAWAREHOUSE was altered.  Delete it and reinsert information
        DELETE FROM ORDERS_DATAWAREHOUSE

        INSERT INTO ORDERS_DATAWAREHOUSE
        Select O_ID, O_D_ID, O_W_ID, O_C_ID, O_ENTRY_D, O_CARRIER_ID, O_OL_CNT, CASE WHEN
O_OL_CNT < 11 THEN 9 ELSE O_ALL_LOCAL END from ORDERS where O_ENTRY_D BETWEEN @StartTime AND
@EndTime
```

```
 --Run the upsert query and capture results from dmv
SET @RunType = 'UPSERT'
exec usp_upsertquery  @runtime = @runtime OUTPUT
SELECT @physical_reads = last_physical_reads from view_dmv where name = @uspsubname
SELECT @logical_reads = last_logical_reads from view_dmv where name = @uspsubname
SELECT @logical_writes = last_logical_writes from view_dmv where name = @uspsubname
SELECT @elapsed_time = last_elapsed_time from view_dmv where name = @uspsubname
insert into _Results_MergeExperiment VALUES (@runID,  @StartTime, @EndTime, @RunType,@runtime,
@elapsed_time, @physical_reads,  @logical_reads, @logical_writes, @TargetInserts,@TargetDeletes,@TargetUpdates)


  FETCH NEXT FROM DateList
   INTO @runID, @StartTime, @EndTime, @StartTime2, @EndTime2
END

CLOSE DateList
DEALLOCATE DateList
GO
/****** Object:  StoredProcedure [dbo].[usp_runDateExperiment]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_runDateExperiment]
/*****************************************
Procedure to run the Date Experiment


*****************************************/
as
set Statistics TIME ON
set Statistics IO ON
set NOCOUNT ON

delete from _Results_DateExperiment
declare @runID int
declare @physical_reads int
declare @physical_writes int
declare @logical_reads int
declare @logical_writes int
declare @elapsed_time int
declare @runtime datetime
declare @RunType nvarchar(50)
--Procedure names retrieve information from the dynamic management views (dmv)
declare @uspname nvarchar(50)= 'usp_daterangequery'
declare @uspsubname nvarchar(50)= 'usp_datepartquery'
declare @StartTime date
declare @EndTime date
declare @MonthPart int
declare @YearPart int
declare @NumRecords int
declare @month datetime
declare DateList cursor for
    --Run the experiment for each row in the date samples table (_dateranges)
    select ID, StartTime, EndTime from dbo._dateranges ORDER BY ID
            OPEN DateList
    FETCH NEXT FROM DateList
    INTO @runID, @StartTime, @EndTime
    WHILE @@FETCH_STATUS = 0
    BEGIN

    set @MonthPart = DATEPART(M, @StartTime)
    select @MonthPart
    set @YearPart = DATEPART(year, @StartTime)
    select @YearPart

 --Run the date range query and capture results from dmv
    SET @RunType  = 'DATEBETWEEN'
    DBCC DROPCLEANBUFFERS
```

```
		--Sets the EndTime to just before midnight on the last day of the month
		set @month= dateadd(day, 1, @EndTime)
		set @month = dateadd(MS, -2, @month)

		exec usp_daterangequery @rangestart = @StartTime, @rangeend = @month , @runtime = @runtime
OUTPUT, @NumRecords = @NumRecords OUTPUT
		SELECT @physical_reads = last_physical_reads from view_dmv where name = @uspname
		SELECT @logical_reads = last_logical_reads from view_dmv where name = @uspname
		SELECT @logical_writes = last_logical_writes from view_dmv where name = @uspname
		SELECT @elapsed_time = last_elapsed_time from view_dmv where name = @uspname
		insert into _Results_DateExperiment VALUES (@runID, @StartTime, @EndTime, @RunType, @NumRecords,
@runtime, @elapsed_time, @physical_reads,  @logical_reads, @logical_writes)

		--Run the date part query and capture results from dmv
		SET @RunType = 'DATEPART'
		DBCC DROPCLEANBUFFERS
		exec usp_datepartquery @datemonth = @MonthPart ,  @dateyear = @YearPart,  @runtime = @runtime
OUTPUT, @NumRecords = @NumRecords OUTPUT
		SELECT @physical_reads = last_physical_reads from view_dmv where name = @uspsubname
		SELECT @logical_reads = last_logical_reads from view_dmv where name = @uspsubname
		SELECT @logical_writes = last_logical_writes from view_dmv where name = @uspsubname
		SELECT @elapsed_time = last_elapsed_time from view_dmv where name = @uspsubname
		insert into _Results_DateExperiment VALUES (@runID,  @StartTime, @EndTime, @RunType,
@NumRecords, @runtime, @elapsed_time, @physical_reads,  @logical_reads, @logical_writes)

				DBCC DROPCLEANBUFFERS
		 FETCH NEXT FROM DateList
		  INTO @runID, @StartTime, @EndTime
END

		CLOSE DateList
		DEALLOCATE DateList
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateSamples]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROC [dbo].[uspPopulateSamples]
@randseed int,
@numSamples int
AS
/*******************************************
Populates the item sample table

*******************************************/
DELETE FROM _itemsample
DECLARE @minvalue int

DECLARE @maxvalue int
DECLARE @itemID int
DECLARE @i int = 0
DECLARE @MaxDistrict int
Declare @MaxWarehouse int
Declare @MaxOrder int

DECLARE @District int
Declare @Warehouse int
Declare @Order int
Declare @Orderline int


SELECT @minvalue =  MIN(ITEM.I_ID) FROM ITEM
SELECT @maxvalue =  MAX(ITEM.I_ID) FROM ITEM
SELECT @MaxDistrict = MAX(D_ID) FROM DISTRICT
SELECT @MaxWarehouse = MAX(W_ID) FROM WAREHOUSE
SELECT @MaxOrder = MAX(O_ID) FROM ORDERS


SELECT RAND(@randseed)
```

```
while (@i < @numSamples)
		BEGIN
				--Pick a random order and a random item from the first five lines of the order
				exec dbo.uspRandomInt 1, @MaxOrder, @RandomInt = @Order OUTPUT
				exec dbo.uspRandomInt 1, 5, @RandomInt = @Orderline OUTPUT
				SELECT TOP 1 @itemID = ORDERLINE.OL_I_ID FROM ORDERLINE WHERE OL_O_ID = @Order
				print @itemID
				--Check to see if the item is already in the sample table
				IF @itemID IS NOT NULL AND @itemID NOT IN (SELECT ItemID FROM _itemsample)
				BEGIN
						set @i = @i + 1
						insert into _itemsample (ID, ItemID) VALUES (@i, @itemID)
				END
END
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateOrderLine]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateOrderLine]
/*****************************************
Inserts Order Lines into the ORDERLINES table
*****************************************/
@numLines int,
@OID int,
@DID int,
@WID int,
@EntryDate datetime

AS

declare @orderlineID int = 0
declare @itemID int
declare @amount numeric(6,2)
declare @DistInfo varchar(24)

		WHILE @orderlineID < @numLines
		BEGIN

		 set @orderlineID = @orderlineID + 1;
						exec dbo.uspRandomPrice 1.00, 9999.99, @RandomPrice = @amount OUTPUT
						exec dbo.uspRandomString 24, 24, 32, 160, @Str = @DistInfo OUTPUT
						exec uspRandomInt 1, 100000, @itemID OUTPUT
								INSERT INTO dbo.ORDERLINE(OL_O_ID,
																OL_D_ID, OL_W_ID,
OL_NUMBER, OL_I_ID, OL_SUPPLY_W_ID, OL_DELIVERY_D, OL_QUANTITY, OL_AMOUNT, OL_DIST_INFO )
								VALUES
								(@OID, @DID, @WID, @orderlineid, @itemid, @WID, @EntryDate, 5,
@amount, @DistInfo )

						END
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateOrder]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateOrder]
/*****************************************
Inserts @numOrdersPerCustomer * number of Customers
into the dbo.ORDERS table.  A deviation from the TPC-C specification
was needed to accomodate the Join Experiment-
O_ID is unique across all orders
*****************************************/
@numOrdersPerCustomer int,
@minorderdate datetime,
@maxorderdate datetime

AS
```

```
--SET Nocount on
declare @numCustomers int
select @numCustomers = MAX(C_ID) FROM dbo.CUSTOMER
declare @randInt int
declare @DID int
declare @WID int
declare @CID int
declare @i int
declare @j int = 0
        declare @entrydate datetime = @minorderdate
        declare @maxdate datetime = @maxorderdate
        declare @basedate datetime = @minorderdate
        declare @secondinterval int
        declare @randomInt int
  set @secondinterval = datediff(second,  @entrydate , @maxdate )
  set @secondinterval = @secondinterval/(@numOrdersPerCustomer)



        declare CustList cursor for
        select D_ID, D_W_ID from dbo.District ORDER BY D_W_ID
                OPEN CustList
        FETCH NEXT FROM CustList
        INTO @DID, @WID
        WHILE @@FETCH_STATUS = 0
        BEGIN

         set @basedate = @minorderdate
         declare @O_CARRIER_ID int = null

         SET @CID  = 1
         declare @orderid int = 1
         while @CID < (@numCustomers + 1)
         begin
                set @i = 0
                set @basedate = @minorderdate
                while @i < @numOrdersPerCustomer
                begin


                 if (@i <2101)
          BEGIN
                exec dbo.uspRandomInt 1, 10, @RandomInt = @O_CARRIER_ID OUTPUT

          END
         declare @O_OL_CNT numeric(2,0)
                exec dbo.uspRandomInt 5, 15, @RandomInt = @O_OL_CNT OUTPUT
         declare @O_ALL_LOCAL numeric (1,0) = 1

         exec uspRandomInt 0, @secondinterval, @randomInt OUTPUT
         set @entrydate= DATEADD(second, @randomInt, @basedate)
         set @j = @j + 1

                        INSERT INTO dbo.ORDERS(O_ID,
                                                                        O_D_ID, O_W_ID, O_C_ID,
O_ENTRY_D, O_CARRIER_ID, O_OL_CNT, O_ALL_LOCAL )
                                VALUES
                                (@j, @DID, @WID, @CID, @entrydate, @O_CARRIER_ID, @O_OL_CNT,
@O_ALL_LOCAL )

                        exec dbo.uspPopulateOrderLine @O_OL_CNT, @j, @DID, @WID, @entrydate

                set @basedate= DATEADD(second, @secondinterval, @basedate)


                SET @orderid = @orderid+1
                SET @i = @i + 1
                end
                SET @CID = @CID + 1
```

```
                    end

                FETCH NEXT FROM CustList
                  INTO @DID, @WID

              END
              CLOSE CustList
              DEALLOCATE CustList
GO
/****** Object:  StoredProcedure [dbo].[usp_runJoinExperiment]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[usp_runJoinExperiment]
as
/*****************************************
Query to run the Join Experiment


******************************************/
set Statistics TIME ON
set Statistics IO ON
set NOCOUNT ON

delete from _Results_JoinExperiment
declare @runID int
declare @physical_reads int
declare @physical_writes int
declare @logical_reads int
declare @logical_writes int
declare @elapsed_time int
declare @runtime datetime
declare @RunType nvarchar(50)
--Procedure names retrieve information from the dynamic management views (dmv)
declare @uspname nvarchar(50)= 'usp_joinquery'
declare @uspsubname nvarchar(50)= 'usp_subquery'
declare @itemid int
declare @NumRecords int
--Run the join experiment on each record in the item sample
declare ItemList cursor for
    select ID, ItemId from dbo._itemsample ORDER BY ID
                    OPEN ItemList
              FETCH NEXT FROM ItemList
              INTO @runID, @itemid
              WHILE @@FETCH_STATUS = 0
              BEGIN
              --Run the join query and capture results from dmv
              SET @RunType  = 'JOIN'
              DBCC DROPCLEANBUFFERS
              exec usp_joinquery @itemid = @itemid, @runtime = @runtime OUTPUT, @NumRecords = @NumRecords
OUTPUT
              SELECT @physical_reads = last_physical_reads from view_dmv where name = @uspname
              SELECT @logical_reads = last_logical_reads from view_dmv where name = @uspname
              SELECT @logical_writes = last_logical_writes from view_dmv where name = @uspname
              SELECT @elapsed_time = last_elapsed_time from view_dmv where name = @uspname
              insert into _Results_JoinExperiment VALUES (@runID, @RunType,@itemid, @NumRecords, @runtime,
@elapsed_time, @physical_reads,  @logical_reads, @logical_writes)

              --Run the subquery and capture results from dmv
              SET @RunType = 'SUBQUERY'
              DBCC DROPCLEANBUFFERS
              exec usp_subquery @itemid = @itemid, @runtime = @runtime OUTPUT, @NumRecords = @NumRecords
OUTPUT
              SELECT @physical_reads = last_physical_reads from view_dmv where name = @uspsubname
              SELECT @logical_reads = last_logical_reads from view_dmv where name = @uspsubname
              SELECT @logical_writes = last_logical_writes from view_dmv where name = @uspsubname
              SELECT @elapsed_time = last_elapsed_time from view_dmv where name = @uspsubname
```

```
        insert into _Results_JoinExperiment VALUES (@runID, @RunType,@itemid, @NumRecords, @runtime,
@elapsed_time, @physical_reads,  @logical_reads, @logical_writes)

         FETCH NEXT FROM ItemList
          INTO @runID, @itemid
END

        CLOSE ItemList
        DEALLOCATE ItemList
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateNUOrder]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateNUOrder]
/******************************************
Inserts @numOrdersPerCustomer * number of Customers
into the dbo.ORDERS table using NonUniform data for itemID and O_ENTRY_D
******************************************/
@numOrdersPerCustomer int,
@minorderdate datetime,
@maxorderdate datetime

AS

declare @numCustomers int
select @numCustomers = MAX(C_ID) FROM dbo.CUSTOMER
declare @randInt int
declare @DID int
declare @WID int
declare @CID int
declare @i int
declare @j int = 0

declare @entrydate datetime = @minorderdate
declare @maxdate datetime = @maxorderdate
declare @basedate datetime = @minorderdate
declare @secondinterval int
declare @randomInt int
set @secondinterval = datediff(second,  @entrydate , @maxdate )
set @secondinterval = @secondinterval/(@numOrdersPerCustomer)

declare @baseyear int
declare @year int
declare @randyear int
--@A and @C as defined in TPC 2.1.6
declare @C int = 250
declare @A int =8191
declare @month int
declare @day int
declare @randDate datetime
select @year = DATEPART(year, @maxorderdate)
select @baseyear = DATEPART(year, @basedate)



        --Loop through the districts
        declare CustList cursor for
        select D_ID, D_W_ID from dbo.District ORDER BY D_W_ID
                OPEN CustList
        FETCH NEXT FROM CustList
        INTO @DID, @WID
        WHILE @@FETCH_STATUS = 0
        BEGIN
         set @basedate = @minorderdate
         declare @O_CARRIER_ID int = null
         --Loop through the customers in each district
         SET @CID  = 1
         declare @orderid int = 1
```

```
        while @CID < (@numCustomers + 1)
        begin
                set @i = 0
                set @basedate = @minorderdate

                select @baseyear = DATEPART(year, @basedate)
                --Generate @numOrdersPerCustomer for each customer
                while @i < @numOrdersPerCustomer
                begin

                        if (@i <2101)
                        BEGIN
                                exec dbo.uspRandomInt 1, 10, @RandomInt = @O_CARRIER_ID OUTPUT
                        END
                        declare @O_OL_CNT numeric(2,0)
                                exec dbo.uspRandomInt 5, 15, @RandomInt = @O_OL_CNT OUTPUT
                        declare @O_ALL_LOCAL numeric (1,0) = 1

                        --Select a random date
                        exec uspRandomInt 0, 12, @month OUTPUT
                        exec uspRandomInt 0, 31, @day OUTPUT
                        exec uspRandomInt @baseyear, @year, @randyear OUTPUT
                        select @entrydate = dateadd(mm,(@randyear-1900)* 12 + @month - 1,0) + (@day-1)

                  set @j = @j + 1

                        INSERT INTO dbo.ORDERS(O_ID,
                                                                        O_D_ID, O_W_ID, O_C_ID,
O_ENTRY_D, O_CARRIER_ID, O_OL_CNT, O_ALL_LOCAL )
                                VALUES
                                (@j, @DID, @WID, @CID, @entrydate, @O_CARRIER_ID, @O_OL_CNT,
@O_ALL_LOCAL )

                        exec dbo.uspPopulateNUOrderLine @O_OL_CNT, @j, @DID, @WID, @entrydate, @A,
@C

                --The date is skewed to more current dates making the non uniform
                set @basedate = @entrydate

                SET @orderid = @orderid+1
                SET @i = @i + 1
                end
                SET @CID = @CID + 1
                 end


        FETCH NEXT FROM CustList
          INTO @DID, @WID

       END
       CLOSE CustList
       DEALLOCATE CustList
GO
/****** Object:  StoredProcedure [dbo].[uspPopulateTPCTables]    Script Date: 04/22/2009 23:12:27 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[uspPopulateTPCTables]
@randseed int = 400,
@numWarehouses int = 10,
@numDistricts int = 10,
@numCustomers int = 100,
@numOrders int = 100,
@mindate datetime = '01/01/1969',
@maxdate datetime = '01/01/2009',
@yearsofdatesamples int = 25,
@span int = 6,
@range int = 3,
@numItemSamples int = 200,
```

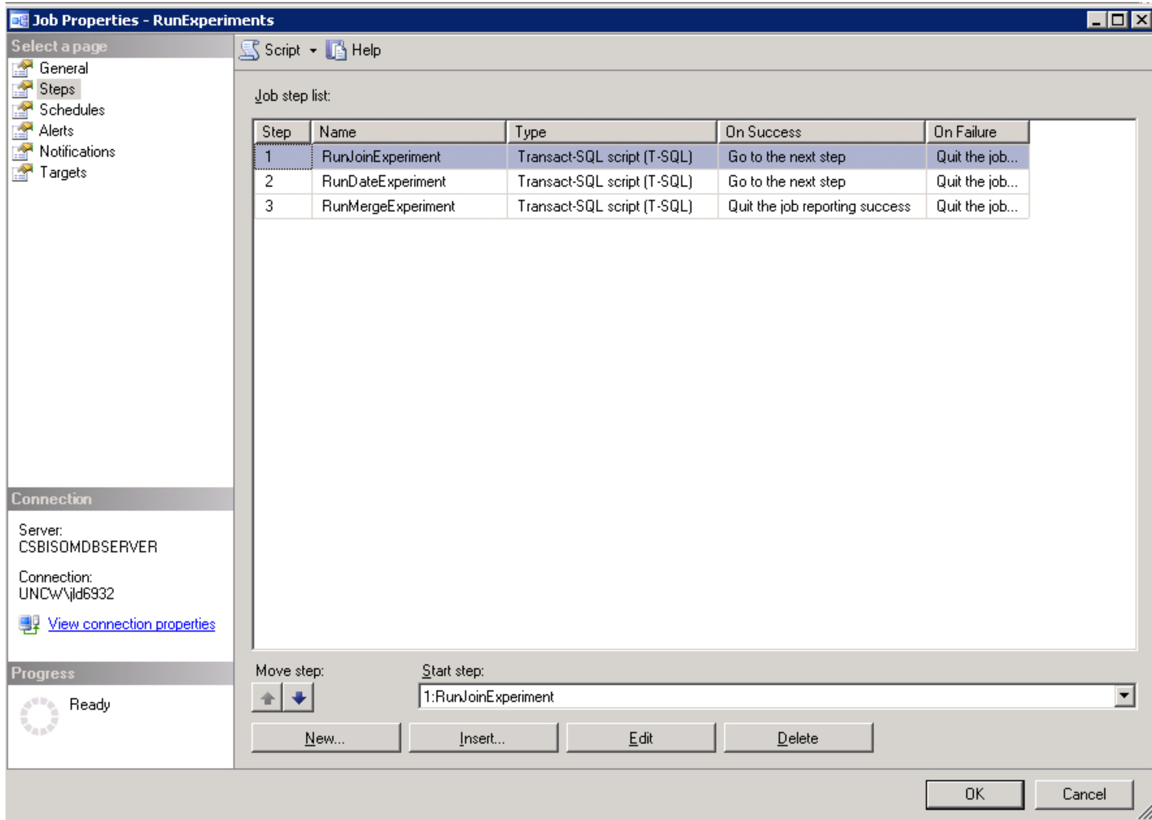
```
@indexing char = 'm',
@NuRand char = 'u'
AS
/*****************************************
Used to generate random data for TPC table.

*****************************************/


SET NOCOUNT ON
--Use seeding to allow duplication of results
select rand(@randseed)
--remove any prior entries from the core tables
DELETE FROM _itemsample
DELETE FROM _dateranges
DELETE FROM _mergequarters
DELETE FROM ORDERLINE
DELETE FROM ORDERS
DELETE FROM CUSTOMER
DELETE FROM DISTRICT
DELETE FROM WAREHOUSE

--Add additional indexing
if @indexing = 'e'
exec _addIndexes
--Add Warehouses
exec uspPopulateWarehouse @numWarehouses
print 'Warehouses Complete'
--Add Districts
exec uspPopulateDistrict  @numDistricts
print 'Districts Complete'
--Add Customers
exec uspPopulateCustomer  @numCustomers
print 'Customers Complete'
--If 'n' option is selected, Add Orders with non-uniform itemIDs and O_Entry_D
if @NuRand = 'n'
exec uspPopulateNUOrder @numOrders, @mindate, @maxdate
--Otherwise, Add Orders with uniform itemIDs and O_Entry_D
else
exec uspPopulateOrder @numOrders, @mindate, @maxdate
print 'Orders Complete'
--Add random samples to the sample tables
exec uspPopulateSamples @randseed, @numItemSamples
exec uspCreateDateDataSet @maxdate, @yearsofdatesamples
exec uspCreateQuartersDataSet @span, @range
print 'Samples Complete'
GO
/****** Object:  ForeignKey [FK_ORDER_merge]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[_mergesource]  WITH CHECK ADD  CONSTRAINT [FK_ORDER_merge] FOREIGN
KEY([O_C_ID], [O_D_ID], [O_W_ID])
REFERENCES [dbo].[CUSTOMER] ([C_ID], [C_D_ID], [C_W_ID])
GO
ALTER TABLE [dbo].[_mergesource] CHECK CONSTRAINT [FK_ORDER_merge]
GO
/****** Object:  ForeignKey [FK_CUSTOMER_DISTRICT]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[CUSTOMER]  WITH CHECK ADD  CONSTRAINT [FK_CUSTOMER_DISTRICT] FOREIGN
KEY([C_D_ID], [C_W_ID])
REFERENCES [dbo].[DISTRICT] ([D_ID], [D_W_ID])
GO
ALTER TABLE [dbo].[CUSTOMER] CHECK CONSTRAINT [FK_CUSTOMER_DISTRICT]
GO
/****** Object:  ForeignKey [FK_DISTRICT_WAREHOUSE]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[DISTRICT]  WITH CHECK ADD  CONSTRAINT [FK_DISTRICT_WAREHOUSE] FOREIGN
KEY([D_W_ID])
REFERENCES [dbo].[WAREHOUSE] ([W_ID])
GO
ALTER TABLE [dbo].[DISTRICT] CHECK CONSTRAINT [FK_DISTRICT_WAREHOUSE]
GO
/****** Object:  ForeignKey [FK_HISTORY_CUSTOMER]    Script Date: 04/22/2009 23:12:29 ******/
```

```
ALTER TABLE [dbo].[HISTORY]  WITH CHECK ADD  CONSTRAINT [FK_HISTORY_CUSTOMER] FOREIGN
KEY([H_D_ID], [H_W_ID])
REFERENCES [dbo].[DISTRICT] ([D_ID], [D_W_ID])
GO
ALTER TABLE [dbo].[HISTORY] CHECK CONSTRAINT [FK_HISTORY_CUSTOMER]
GO
/****** Object:  ForeignKey [FK_NEW-ORDER_ORDER]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[NEW-ORDER]  WITH CHECK ADD  CONSTRAINT [FK_NEW-ORDER_ORDER] FOREIGN
KEY([NO_O_ID], [NO_D_ID], [NO_W_ID])
REFERENCES [dbo].[ORDERS] ([O_ID], [O_D_ID], [O_W_ID])
GO
ALTER TABLE [dbo].[NEW-ORDER] CHECK CONSTRAINT [FK_NEW-ORDER_ORDER]
GO
/****** Object:  ForeignKey [FK_ORDER-LINE_ORDER]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[ORDERLINE]  WITH CHECK ADD  CONSTRAINT [FK_ORDER-LINE_ORDER] FOREIGN
KEY([OL_O_ID], [OL_D_ID], [OL_W_ID])
REFERENCES [dbo].[ORDERS] ([O_ID], [O_D_ID], [O_W_ID])
GO
ALTER TABLE [dbo].[ORDERLINE] CHECK CONSTRAINT [FK_ORDER-LINE_ORDER]
GO
/****** Object:  ForeignKey [FK_ORDER_ORDER]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[ORDERS]  WITH CHECK ADD  CONSTRAINT [FK_ORDER_ORDER] FOREIGN KEY([O_C_ID],
[O_D_ID], [O_W_ID])
REFERENCES [dbo].[CUSTOMER] ([C_ID], [C_D_ID], [C_W_ID])
GO
ALTER TABLE [dbo].[ORDERS] CHECK CONSTRAINT [FK_ORDER_ORDER]
GO
/****** Object:  ForeignKey [FK_ORDER_ORDERWAREHOUSE]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[ORDERS_DATAWAREHOUSE]  WITH CHECK ADD  CONSTRAINT
[FK_ORDER_ORDERWAREHOUSE] FOREIGN KEY([O_C_ID], [O_D_ID], [O_W_ID])
REFERENCES [dbo].[CUSTOMER] ([C_ID], [C_D_ID], [C_W_ID])
GO
ALTER TABLE [dbo].[ORDERS_DATAWAREHOUSE] CHECK CONSTRAINT [FK_ORDER_ORDERWAREHOUSE]
GO
/****** Object:  ForeignKey [FK_STOCK_ITEM]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[STOCK]  WITH CHECK ADD  CONSTRAINT [FK_STOCK_ITEM] FOREIGN KEY([S_I_ID])
REFERENCES [dbo].[ITEM] ([I_ID])
GO
ALTER TABLE [dbo].[STOCK] CHECK CONSTRAINT [FK_STOCK_ITEM]
GO
/****** Object:  ForeignKey [FK_STOCK_WAREHOUSE]    Script Date: 04/22/2009 23:12:29 ******/
ALTER TABLE [dbo].[STOCK]  WITH CHECK ADD  CONSTRAINT [FK_STOCK_WAREHOUSE] FOREIGN
KEY([S_W_ID])
REFERENCES [dbo].[WAREHOUSE] ([W_ID])
GO
ALTER TABLE [dbo].[STOCK] CHECK CONSTRAINT [FK_STOCK_WAREHOUSE]
GO
```


Job Properties - RunExperiments
Select a page
General
Steps
Schedules
Alerts
Notifications
Targets
Script
Help
Job step list:
Step | Name | Type | On Success | On Failure
1 | RunJoinExperiment | Transact-SQL script (T-SQL) | Go to the next step | Quit the job...
2 | RunDateExperiment | Transact-SQL script (T-SQL) | Go to the next step | Quit the job...
3 | RunMergeExperiment | Transact-SQL script (T-SQL) | Quit the job reporting success | Quit the job...
Connection
Server:
CSBISOMDBSERVER
Connection:
UNCW\jld6932
View connection properties
Progress
Ready
Move step:
Start step:
1:RunJoinExperiment
New...
Insert...
Edit
Delete
OK
Cancel