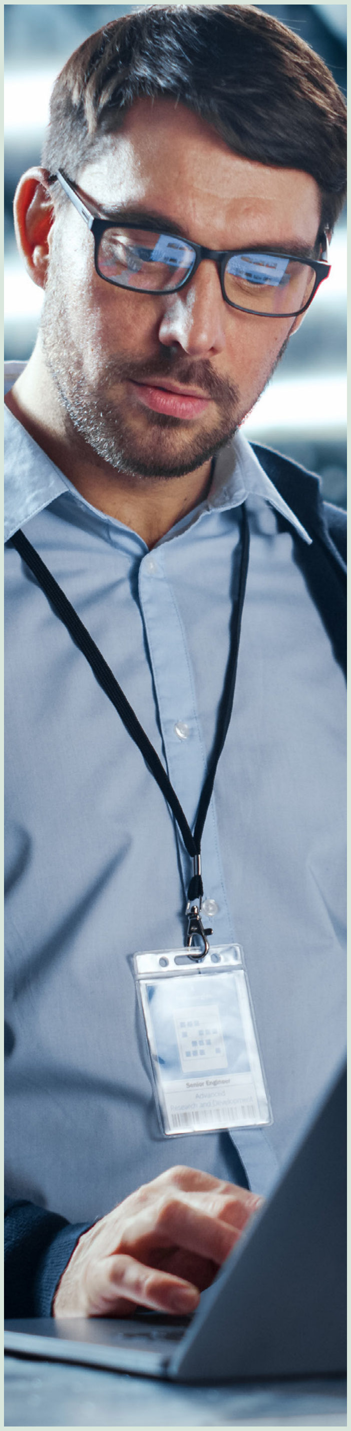
WHITE PAPER



MINIMALISTIC OLTP DATABASES

Abstract

OLTP database can become unwieldy, due to many unused and poorly designed database objects. The reasons for unwieldiness can be unused tables/indexes, duplicate tables/indexes, unnecessary wider datatypes/wider data lengths being assigned to columns, unnecessary audit columns etc. Bulkier OLTP database leads to many operational problems like more IO/memory footprint, bulkier backups, longer duration for housekeeping activities, longer duration for RPO/RTO etc. Minimalistic OLTP database will make working with it easier and its maintenance pleasure. Some of the characteristics of minimalistic databases are normalized tables, necessary columns, narrow datatypes, narrow indexes, minimal IO/memory footprint etc. It will lead to happy usage experience for customers, DBA, sysadmin and development team.

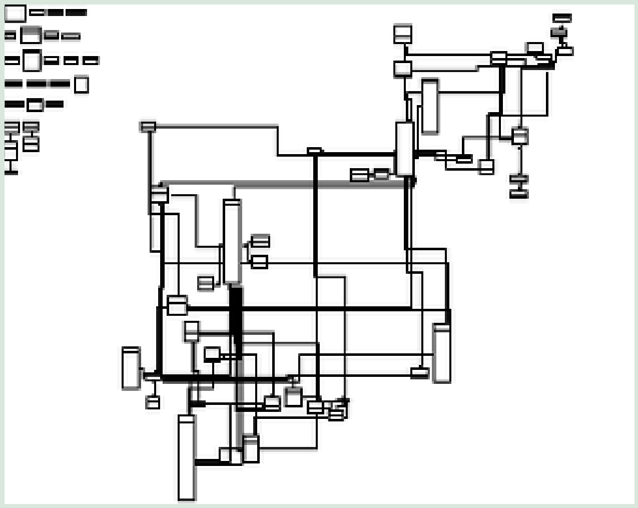
Infosys®
Navigate your next

Cluttered database is like a cluttered house

Cluttered House

- Lots of household objects of different types in house
- With too many objects, the house is cluttered and difficult to maintain.
- Too many objects in the house require more space, money, time to consume, maintain it.
- If a member of house needs a specific object, it requires extra effort to scan through clutter to pick up object
- Not enough storage to add objects; home must be renovated to get extra storage
- On relocation, all the heavy user objects must be moved to the new place
- With duplicate objects, there is additional effort in storing, maintaining them
- With bulkier objects, it is difficult to take them out and put it back
- Lots of unused objects taking space without adding value to house members
- With small items scattered between larger items, it is difficult to walk past and pick up the small items

Cluttered Database



- Lots of user objects of different types in database
- With too many user objects, the database becomes cluttered and difficult to maintain.
- Too many user objects in the database require more space, money, time to consume, maintain it.
- If database user needs a specific data, storage engine has to do extra physical scan through pages and pick up the data
- Not enough storage to add objects; database storage must be scaled up to get extra storage.
- On restoring, all the heavy user objects must be backed up and restored in new server
- With duplicate objects, there is additional effort in storing, maintaining them
- With bulky tables/indexes, it is difficult to physically read them from disk and bring it to memory and again write them back to disk
- Lots of unused objects, taking space, without adding value to database users
- With small datatype values being stored among bigger datatype values, it is difficult to physically scan past them and read the smaller datatype values

Root causes for Database Clutter & Solution to address them

Unused Tables

Unused Tables in the database occupy space without adding value. They also make the backup time longer and restore time longer. For the application, the RTO (Recovery Time Objective) and RPO (Recovery Point Objective) will get increased due to bulkier database. Also, in cases of space constraints for database growth, they pose a serious threat to sudden application unresponsiveness.

Solution: Unused tables must be watched and removed after careful consideration.

Unused Indexes

Unused indexes are worse than unused tables, as they warrant regular maintenance, as part of database housekeeping activities. If there are unused indexes in an active table, it makes the DML transactions longer and as indexes must be in sync with DML operations. When the transactions become longer, there will be longer locks, lock escalations, blocking, connection/query timeout, deadlock issues occurring in the database. The end users will frequently get time out errors and lose trust with the application.

Solution: Unused indexes must be watched for and removed, if they are not adding any value to the application.

Duplicate Tables

Duplicate Tables are like unused tables, taking up space. They could have originally been part and parcel of application. But, with application changes, they might not be in use anymore. They are not adding value to the application and simply taking up space.

Solution: Duplicate tables must be examined and dropped, if we don't need them anymore.

Duplicate Indexes

Duplicate indexes are worse than duplicate tables, as they warrant regular maintenance, as part of database housekeeping activities. Databases will not allow duplicate indexes with same name. But, duplicate indexes with different names can be created for same set of columns. Duplicate indexes don't help in application read queries. Database Optimizer chooses the latest index with updated statistics. So, original index simply remains in the database causing additional maintenance headache. Duplicate indexes are like unused indexes in a table.

Solution: Duplicate indexes based on column list and column order, must be identified and dropped.

Denormalized Columns

Denormalized columns simply duplicate the column data from another table. They can be called as duplicate columns. Normalized database design ensures that there is no insert, update, delete anomalies. Denormalization leads to longer transactions and concurrency issues.

Solution: Unneeded denormalized columns should be removed to keep the database in normalized design. Denormalization should be employed only in rare scenarios to improve read performance and should be carefully handled to avoid DML anomalies.

Wider Datatypes

Wider datatypes lead to more byte storage for columns and lead to more disk IO/memory IO footprint and can lead to disk IO/memory bottlenecks and latency issues. Datatype should be decided based on need and future growth prospect. If there is no clarity on future needs, datatype should be designed based on current requirement. E.g., For age column, TINYINTEGER (range from 0 to 255) is enough. TINYINTEGER is 1 byte storage. If we go for wider datatypes, say BIGINTEGER (8 bytes) for age column, then we are unnecessarily wasting additional 7 bytes in the age column. Below tabulation lists the wastage for age column.

No. of rows	Additional storage due to BIGINTEGER	Additional IO every time in READ/WRITE
10M	7 bytes	70 MB
100M	7 bytes	700 MB or 0.7 GB

Solution: Every column should carefully be analyzed for right datatype and alter datatype if required. Wider datatypes should be allocated only in cases, where we really need wider datatype. E.g., BIGINTEGER datatype for surrogate key column of an ecommerce sales transaction table.

Unicode Datatypes

For ASCII alphanumeric characters storage, we don't need unicode datatypes. Unicode datatypes require double byte storage. So, if we want to store alphabet 'A', in unicode character datatype, it requires twice the amount of storage compared to alphabet 'A' in ASCII character datatype. Additional storage leads to more IO/memory footprint, concurrency issues. E.g., We are having FirstName column, and its length is 255 bytes. We are just going to store English first names. If we use Unicode datatype, with double the byte storage, it will take 510 bytes. Additional 255 bytes needed. The below tabulation lists the wastage for FirstName column.

No. of rows	Additional Storage due to Unicode	Additional IO every time in READ/WRITE
10M	255 bytes	2.5 GB
100M	255 bytes	25 GB

Solution: Every column should be analyzed if we really need unicode datatype. If we don't need unicode datatypes, they should be altered to ASCII equivalent.

Too many variable length columns in a table

Too many variable length columns make row wider. Variable length datatypes save space, when we use only part of the storage. But, they also bring in additional storage in the form of variable block to store the count of variable datatypes columns and variable datatypes column value offset information. In RDBMS systems, when the variable length columns are updated, it can lead to row chaining, as data overflows from the INROWDATA blocks to ROWOVERFLOW blocks. It causes additional IO overhead to bring the overflow data with additional hops when data is read. Also, it causes additional pointer storage to point to the location of row overflow storage.

Solution: Choose variable length datatypes based on need. Go for fixed length datatypes as much as possible. If there are many variable length columns, it is better to go with vertical partitioning of table to avoid row overflowing.

Too many wide LOB data types in a table

If a table has too many wide LOB datatypes and causing data to overflow the INROWDATA portion, there is additional IO overhead to read the LOB data. Also, there is additional pointer storage to point to LOB data.

Solution: Choose LOB datatypes based on need. If there are many LOB columns, it is better to go with vertical partitioning of table to avoid row overflowing.

Lots of Nullable Columns

If a column is nullable, it is anyway going to take space in the case of fixed length datatypes. In case of variable length datatypes, there will not be any storage for data, but still need storage to indicate that the value is null. Null columns lead to more storage, even if the column is not having value.

Solution: Every column needs to be analyzed if we need it. If there is a possibility of so many null values, we need to analyze the need to store it. If null value columns are more, we can try sparse columns to save IO, read performance.

Unused Futuristic Columns

These columns might not be needed currently, but we are defining them for future use. These columns are going to take space and will cause additional IO.

Solution: Every column needs to be analyzed if we really need it. If it is for future use, we need to consider if they can be removed.

Persisted Calculated Columns

Persisted calculated columns are also a kind of duplicate columns only. If a column can be derived from existing columns, storing the derived column will take additional storage, which is completely avoidable. Below is an example of calculated persisted column in SQL SERVER, which leads to additional storage.

```
CREATE TABLE dbo.Employee (
        EmployeeID INT IDENTITY(1, 1) NOT NULL
        ,FirstName VARCHAR(255)
        ,LastName VARCHAR(255)
        ,FullName AS FirstName + ' ' + LastName PERSISTED
        );
```

Solution: Every persisted calculated column needs to be analyzed if we really need it. We can drop these columns and calculate as part of presentation layer needs. These columns should be kept as persisted, only if there is a read performance SLA to satisfy.

Unwanted Historical snapshot/Audit Tables

Keeping Point in time backup of tables is complete waste of space. These historical snapshot tables are not being used. These are like unused tables. Similarly, to track the table changes, we could be using features like Change Data Capture (CDC), Temporal Tables, Change Tracking, Audit tables etc.

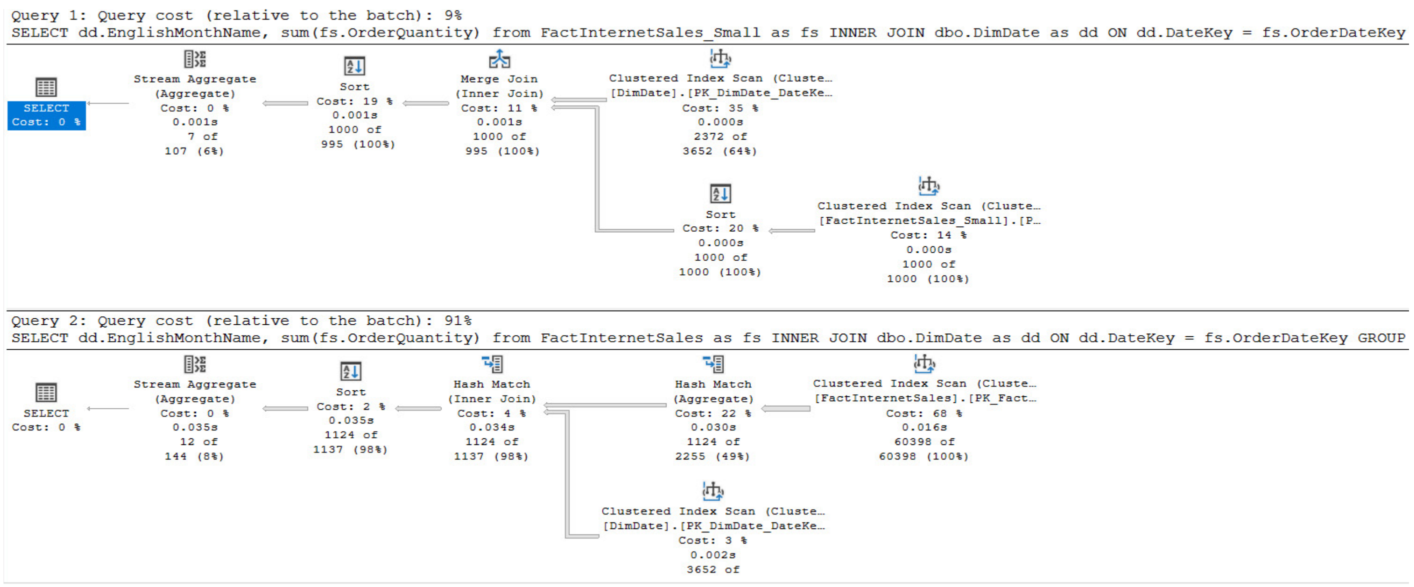
Solution: Every historical table needs to be analyzed, if we need it anymore. Tracking tables/Audit tables need to be analyzed, if they can be removed.

Unwanted historical data outside application usage date range

Keeping historical data in the same table outside the application usage date range, leads to waste of storage space. Based on the purge and archival policy, historical data can be archived first for specific set of years. Archived data should be purged after some years. Keeping historical data on the same table leads to wrong choice of JOIN operator by optimizer and leads to poor performance. Also, in the case of cloud storage, warm and cold storage have different charges and money can be saved. Maintaining and making changes to indexes on huge tables will demand more disk IO/Memory and can bring system to standstill.

In the below scenario, in SQL Server, FactInternetSales_Small is copy of FactInternetSales with 17% of data. The total cost of the query is getting reduced, as it is going for a smaller number of rows and better JOIN operator MERGE JOIN is chosen by the optimizer.

```
SELECT dd.EnglishMonthName
        ,sum(fs.OrderQuantity)
FROM FactInternetSales_Small AS fs
INNER JOIN dbo.DimDate AS dd ON dd.DateKey = fs.OrderDateKey
GROUP BY dd.EnglishMonthName
        ,dd.MonthNumberOfYear
ORDER BY dd.MonthNumberOfYear
SELECT dd.EnglishMonthName
        ,sum(fs.OrderQuantity)
FROM FactInternetSales AS fs
INNER JOIN dbo.DimDate AS dd ON dd.DateKey = fs.OrderDateKey
GROUP BY dd.EnglishMonthName
        ,dd.MonthNumberOfYear
ORDER BY dd.MonthNumberOfYear
```



```
Query 1: Query cost (relative to the batch): 9%
SELECT dd.EnglishMonthName, sum(fs.OrderQuantity) from FactInternetSales_Small as fs INNER JOIN dbo.DimDate as dd ON dd.DateKey = fs.OrderDateKey

SELECT       Stream Aggregate   Sort          Merge Join     Clustered Index Scan (Cluste…
Cost: 0 %    (Aggregate)        Cost: 19 %    (Inner Join)   [DimDate].[PK_DimDate_DateKe…
             Cost: 0 %          0.001s        Cost: 11 %     Cost: 35 %
             0.001s             1000 of       0.001s         0.000s
             7 of               995 (100%)    1000 of        2372 of
             107 (6%)                         995 (100%)     3652 (64%)

                                                             Sort           Clustered Index Scan (Cluste…
                                                             Cost: 20 %     [FactInternetSales_Small].[P…
                                                             0.000s         Cost: 14 %
                                                             1000 of        0.000s
                                                             1000 (100%)    1000 of
                                                                            1000 (100%)

Query 2: Query cost (relative to the batch): 91%
SELECT dd.EnglishMonthName, sum(fs.OrderQuantity) from FactInternetSales as fs INNER JOIN dbo.DimDate as dd ON dd.DateKey = fs.OrderDateKey GROUP

SELECT       Stream Aggregate   Sort          Hash Match     Hash Match     Clustered Index Scan (Cluste…
Cost: 0 %    (Aggregate)        Cost: 2 %     (Inner Join)   (Aggregate)    [FactInternetSales].[PK_Fact…
             Cost: 0 %          0.035s        Cost: 4 %      Cost: 22 %     Cost: 68 %
             0.035s             1124 of       0.034s         0.030s         0.016s
             12 of              1137 (98%)    1124 of        1124 of        60398 of
             144 (8%)                         1137 (98%)     2255 (49%)     60398 (100%)

                                                             Clustered Index Scan (Cluste…
                                                             [DimDate].[PK_DimDate_DateKe…
                                                             Cost: 3 %
                                                             0.002s
                                                             3652 of
```

Solution: Every table needs to be analyzed for date range to keep the data. Data outside the date range has to be archived and later purged, as per archival policy.

Lack of Normalization

If the database is not properly normalized, tables will be wide with duplicate columns. It can also lead to insert/update/delete anomalies.

Solution: Existing data model should at least satisfy third normal form.

Wrong Index Keys Selection

When Index key columns are nullable columns, we need byte still for null values. If the index key column is a non-unique column, uniquifier must be added to make it unique. All these lead to more disk IO/memory footprint.

Solution: Going for narrow, unique, non-null columns will lead to better index performance. Indexes need to be revised for column list and column order. More selective columns should come before less selective columns for better index performance.

Wide Clustering Key for non-clustered index

In SQL Server, every non-clustered index stores the clustering key in the leaf page to traverse back to the data page. Keeping wider clustering key will lead to more IO in index pages and longer transactions.

Solution: In SQL Server, existing clustering key needs to be revisited to go for narrow, unique, non-null, incremental clustering key to reduce the IO related to clustering key. For other RDBMS systems, we need to see how indexes traverse to data pages and handle accordingly.

Heap Tables

In SQL Server RDBMS, heap tables lead to random IO. During UPDATE operations, there will be more fragmentation happening in the heap and it will lead to more IO. Better to have clustered index on the tables to make the IO sequential.

Solution: In SQL Server, it is always preferred to go for clustered index-based table. For other RDBMS systems, heap tables must be examined for random IO reduction.

Characteristics of minimalistic database

- **Necessary tables:** Database will have minimal set of tables to satisfy the application requirements
- **Necessary columns:** Necessary tables will have only the columns which are needed to satisfy application requirements and auditing requirements.
- **Necessary indexes:** Indexes will be minimal and consolidated to satisfy more workloads, instead of separate index for each workload.
- **Normalized design:** The database will be normalized to at least third normal form and DML transactions will be short and fast.
- **Narrow datatypes:** The narrowest datatype to satisfy the column requirement will be chosen. Unicode datatype is chosen, only based on the need.
- **Proper variable length datatypes:** Variable datatypes will also have only sufficient length to ensure that they generally don't overflow the page.
- **Proper nullability:** Null columns are chosen where there is a valid use case.
- **Proper audit columns:** Audit columns are chosen based on the need.
- **Proper history tables with purging/archival:** Proper data purging/archival policy is established and there are regular jobs to take care of the archival/purging tasks.
- **Necessary Audit tables:** Audit tables will be kept minimal and with minimal set of columns.
- **Narrow clustering key:** Clustering key is narrow, non-null, unique and incremental to avoid fragmentation issues and puts limited load on the non-clustered indexes.
- **Vertical partitioning for tables:** Tables are narrow and row lengths are narrow and tables are vertically partitioned

Advantages of minimalistic database

- **Less Disk IO:** As Data is stored efficiently, there is less disk IO for DML operations
- **Less memory IO:** As Data is stored efficiently, less memory IO for DML operations
- **Less concurrency issues:** As transactions are faster, there are less concurrency issues
- **Faster response times:** Due to less disk IO/memory footprint, read operations are faster
- **Faster backups:** As database is holding only the necessary data, backups are faster.
- **Faster RTO/RPO:** With backups being smaller, recovery is fast with less RTO, RPO
- **Disk space saved:** With only necessary data, lots of space is available for database growth
- **Faster logging:** With only necessary data, logging operation is faster

How to design minimalistic database

- **Go for step-by-step database design:** Conceptual model to logical model to physical model to avoid unnecessary columns, tables cropping up and wrong datatypes being assigned
- **Go for normalized database design:** At least, database should be normalized to third normal form.
- **Avoid denormalization as much as possible:** Denormalization leads to duplication and DML anomalies.
- **Go for right-sized datatype:** For every column, analyze the business domain for the possible set of values and choose right-sized datatype accordingly
- **Go for variable datatype on need:** For every column, variable length datatype should be used based on need. Go for fixed length datatype as much as possible.
- **Specify Nullability accurately:** For every column, nullability should be decided after carefully considering business domain. Too many nullable columns signify that the column might not be needed.
- **Workloads based indexes:** Indexes should be created based on workload needs
- **Necessary Audit columns:** Createdby, ModifiedBy, CreatedAt, ModifiedAt are the important audit columns. Other columns should be carefully chosen based on the need.
- **Go for vertical partitioning for wider tables:** For the case of wider tables, think vertically partitioning the table to make the tables narrower and IO footprint lesser.
- **Go for clear data retention policies:** With clear data retention policies, define regular housekeeping jobs for archival, purging
- **Define regular database maintenance jobs:** Have regular database maintenance jobs for Index reorganize, rebuild, statistics update to avoid fragmentation issues
- **Define granular security:** Ensure that only privileged users can create objects in the database

About the Author

Venkataraman Ramasubramanian
Senior Technology Architect

About the Mentor

SeshaSai Koduri
Senior Principal Technology Architect

References

SQL Server: Why Physical Database Design Matters | Pluralsight

For more information, contact askus@infosys.com

Infosys®
Navigate your next

Infosys.com | NYSE: INFY

Stay Connected