Analysis and comparing the physical storage structures and types

Sport & Tourism, Fitness

First of all define comparative framework. Recommend one product for organizations of around 2000-4000 employees with sound reasoning based on Physical Storage Structures Introduction to Physical Storage Structures One characteristic of an RDBMS is the independence of logical data structures such as tables, views, and indexes from physical storage structures.\n

Because physical and logical structures are separate, you can manage physical storage of data without affecting access to logical structures. For example, renaming a database file does not rename the tables stored in it. The following sections explain the physical database structures of an Oracle database, including datafiles, redo log files, and control files. Datafiles Every Oracle database has one or more physical datafiles. The datafiles contain all the database data. The data of logical database structures, such as tables and indexes, is physically stored in the datafiles allocated for a database.

\n

The characteristics of datafiles are:

\n\n

\n \t

- A datafile can be associated with only one database. \n \t
- Datafiles can have certain characteristics set to let them automatically extend when the database runs out of space. \n \t
- One or more datafiles form a logical unit of database storage called a tablespace. \n

\n

Data in a datafile is read, as needed, during normal database operation and stored in the memory cache of Oracle. For example, assume that a user wants to access some data in a table of a database. If the requested information is not already in the memory cache for the database, then it is read from the appropriate atafiles and stored in memory. Modified or new data is not necessarily written to a datafile immediately. To reduce the amount of disk access and to increase performance, data is pooled in memory and written to the appropriate datafiles all at once, as determined by the database writer process (DBWn) background process. Control Files Every Oracle database has a control file. A control file contains entries that specify the physical structure of the database. For example, it contains the following information:

\n\n

\n \t

- Database name \n \t
- Names and locations of datafiles and redo log files \n \t
- Time stamp of database creation \n

\n

Oracle can multiplex the control file, that is, simultaneously maintain a number of identical control file copies, to protect against afailureinvolving the control file. Every time an instance of an Oracle database is started, its control file identifies the database and redo log files that must be opened for

database operation to proceed. If the physical makeup of the database is altered, (for example, if a new datafile or redo log file is created), then the control file is automatically modified by Oracle to reflect the change. A control file is also used in database recovery. Redo Log Files

\n

Every Oracle database has a set of two or more redo log files. The set of redo log files is collectively known as the redo log for the database. A redo log is made up of redo entries (also called redo records). The primary function of the redo log is to record all changes made to data. If a failure prevents modified data from being permanently written to the datafiles, then the changes can be obtained from the redo log, so work is never lost. To protect against a failure involving the redo log itself, Oracle allows a multiplexed redo log so that two or more copies of the redo log can be maintained on different disks.

\n

The information in a redo log file is used only to recover the database from a system or media failure that prevents database data from being written to the datafiles. For example, if an unexpected power outage terminates database operation, then data in memory cannot be written to the datafiles, and the data is lost. However, lost data can be recovered when the database is opened, after power is restored. By applying the information in the most recent redo log files to the database datafiles, Oracle restores the database to the time at which the power failure occurred.

\n

The process of applying the redo log during a recovery operation is called rolling forward. Archive Log Files You can enable automatic archiving of the redo log. Oracle automatically archives log files when the database is in ARCHIVELOG mode. Parameter Files Parameter files contain a list of configuration parameters for that instance and database. Oracle recommends that you create a server parameter file (SPFILE) as a dynamic means of maintaining initialization parameters. A server parameter file lets you store and manage your initialization parameters persistently in a server-side disk file.

\n

Alert and Trace Log Files Each server and background process can write to an associated trace file. When an internal error is detected by a process, it dumps information about the error to its trace file. Some of the information written to a trace file is intended for the database administrator, while other information is for Oracle Support Services. Trace file information is also used to tune applications and instances. The alert file, or alert log, is a special trace file. The alert file of a database is a chronological log of messages and errors. Backup Files To restore a file is to replace it with a backup file.

\n

Typically, you restore a file when a media failure or user error has damaged or deleted the original file. User-managed backup and recovery requires you to actually restore backup files before you can perform a trial recovery of the

backups. Server-managed backup and recovery manages the backup process, such as scheduling of backups, as well as the recovery process, such as applying the correct backup file when recovery is needed. A database instance is a set of memory structures that manage database files. Figure 11-1 shows the relationship between the instance and the files that it manages.

\n

Figure 11-1 Database Instance and Database Files Mechanisms for Storing Database Files Several mechanisms are available for allocating and managing the storage of these files. The most common mechanisms include: 1. Oracle Automatic Storage Management (Oracle ASM) Oracle ASM includes a file system designed exclusively for use by Oracle Database. 2. Operating system file system Most Oracle databases store files in a file system, which is a data structure built inside a contiguous disk address space. All operating systems have file managers that allocate and deallocate disk space into files within a file system.

\n

A file system enables disk space to be allocated to many files. Each file has a name and is made to appear as a contiguous address space to applications such as Oracle Database. The database can create, read, write, resize, and delete files. A file system is commonly built on top of a logical volume constructed by a software package called a logical volume manager (LVM). The LVM enables pieces of multiple physical disks to be combined into

a single contiguous address space that appears as one disk to higher layers of software. 3. Raw device Raw devices are disk partitions or logical volumes not formatted with a file system.

\n

The primary benefit of raw devices is the ability to perform direct I/O and to write larger buffers. In direct I/O, applications write to and read from the storage device directly, bypassing the operating system buffer cache. 4. Cluster file system A cluster file system is software that enables multiple computers to share file storage while maintaining consistent space allocation and file content. In an Oracle RACenvironment, a cluster file system makes shared storage appears as a file system shared by many computers in a clustered environment.

\n

With a cluster file system, the failure of a computer in the cluster does not make the file system unavailable. In an operating system file system, however, if a computer sharing files through NFS or other means fails, then the file system is unavailable. A database employs a combination of the preceding storage mechanisms. For example, a database could store the control files and online redo log files in a traditional file system, some user data files on raw partitions, the remaining data files in Oracle ASM, and archived the redo log files to a cluster file system. Indexes in Oracle

\n

There are several types of indexes available in Oracle all designed for different circumstances:

\n\n

\n \t

1. b*tree indexes - the most common type (especially in OLTP environments) and the default type \n \t
2. b*tree cluster indexes - for clusters \n \t
3. hash cluster indexes - for hash clusters \n \t
4. reverse key indexes - useful in Oracle Real Application Cluster (RAC) applications \n \t
5. bitmap indexes - common in data warehouse applications \n \t
6. partitioned indexes - also useful for data warehouse applications \n \t
7. function-based indexes \n \t
8. index organized tables \n \t
9. domain indexes \n

\n

Let's look at these Oracle index types in a little more detail. B*Tree Indexes B*tree stands for balanced tree. This means that the height of the index is the same for all values thereby ensuring that retrieving the data for any one value takes approximately the same amount of time as for any other value. Oracle b*tree indexes are best used when each value has a high cardinality (low number of occurrences)for example primary key indexes or unique indexes. One important point to note is that NULL values are not indexed.

They are the most common type of index in OLTP systems. B*Tree Cluster Indexes

\n

These are B*tree index defined for clusters. Clusters are two or more tables with one or more common columns and are usually accessed together (via a join). CREATE INDEX product_orders_ix ON CLUSTER product_orders; Hash Cluster Indexes In a hash cluster rows that have the same hash key value (generated by a hash function) are stored together in the Oracle database. Hash clusters are equivalent to indexed clusters, except the index key is replaced with a hash function. This also means that here is no separate index as the hash is the index. CREATE CLUSTER emp_dept_cluster (dept_id NUMBER) HASHKEYS 50; Reverse Key Indexes

\n

These are typically used in Oracle Real Application Cluster (RAC) applications. In this type of index the bytes of each of the indexed columns are reversed (but the column order is maintained). This is useful when new data is always inserted at one end of the index as occurs when using a sequence as it ensures new index values are created evenly across the leaf blocks preventing the index from becoming unbalanced which may in turn affect performance. CREATE INDEX emp_ix ON emp(emp_id) REVERSE; Bitmap Indexes These are commonly used in data warehouse applications for tables with no updates and whose columns have low cardinality (i. . there are few distinct values). In this type of index Oracle stores a bitmap for each

distinct value in the index with 1 bit for each row in the table. These bitmaps are expensive to maintain and are therefore not suitable for applications which make a lot of writes to the data. For example consider a car manufacturer which records information about cars sold including the colour of each car. Each colour is likely to occur many times and is therefore suitable for a bitmap index. CREATE BITMAP INDEX car_col ON cars(colour) REVERSE; Partitioned Indexes

\n

Partitioned Indexes are also useful in Oracle datawarehouse applications where there is a large amount of data that is partitioned by a particular dimension such as time. Partition indexes can either be created as local partitioned indexes or global partitioned indexes. Local partitioned indexes mean that the index is partitioned on the same columns and with the same number of partitions as the table. For global partitioned indexes the partitioning is user defined and is not the same as the underlying table. Refer to the create index statement in the Oracle SQL language reference for details. Function-based Indexes

\n

As the name suggests these are indexes created on the result of a function modifying a column value. For example CREATE INDEX upp_ename ON emp(UPPER(ename((; The function must be deterministic (always return the same value for the same input). Index Organized Tables In an index-organized table all the data is stored in the Oracle database in a B*tree index

structure defined on the table's primary key. This is ideal when related pieces of data must be stored together or data must be physically stored in a specific order. Index-organized tables are often used for information retrieval, spatial and OLAP applications.

\n

Domain Indexes These indexes are created by user-defined indexing routines and enable the user to define his or her own indexes on custom data types (domains) such as pictures, maps or fingerprints for example. These types of index require in-depth knowledge about the data and how it will be accessed. Indexes in Sql Server Index type| Description| Clustered| A clustered index sorts and stores the data rows of the table or view in order based on the clustered index key. The clustered index is implemented as a B-tree index structure that supports fast retrieval of the rows, based on their clustered index key values. Nonclustered| A nonclustered index can be defined on a table or view with a clustered index or on a heap. Each index row in the nonclustered index contains the nonclustered key value and a row locator. This locator points to the data row in the clustered index or heap having the key value. The rows in the index are stored in the order of the index key values, but the data rows are not guaranteed to be in any particular order unless a clustered index is created on the table. | Unique| A unique index ensures that the index key contains no duplicate values and therefore every row in the table or view is in some way unique.

\n

Both clustered and nonclustered indexes can be unique. | Index with included columns| A nonclustered index that is extended to include nonkey columns in addition to the key columns. | Full-text| A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server. It provides efficient support for sophisticated word searches in character string data. | Spatial| A spatial index provides the ability to perform certain operations more efficiently on spatial objects (spatial data) in a column of the geometry data type.

\n

The spatial index reduces the number of objects on which relatively costly spatial operations need to be applied. | Filtered| An optimized nonclustered index especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes. | XML| A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the xml data type column. | SQL Server Storage Structures

\n

SQL Server does not see data and storage in exactly the same way a DBA or end-user does. DBA sees initialized devices, device fragments allocated to databases, segments defined within Databases, tables defined within segments, and rows stored in tables. SQL Server views storage at a lower

level as device fragments allocated to databases, pages allocated to tables and indexes within the database, and information stored on pages. There are two basic types of storage structures in a database. * Linked data pages * Index trees. All information in SQL Server is stored at the page level. When a database is created, all space

\n

Allocated to it is divided into a number of pages, each page 2KB in size. There are five types of pages within SQL Server: 1. Data and log pages 2. Index pages 3. Text/image pages 4. Allocation pages 5. Distribution pages All pages in SQL Server contain a page header. The page header is 32 bytes in size and contains the logical page number, the next and previous logical page numbers in the page linkage, the object_id of the object to which the page belongs, the minimum row size, the next available row number within the page, and the byte location of the start of the free space on the page.

\n

The contents of a page header can be examined by using the dbcc page command. You must be logged in as sa to run the dbcc page command. The syntax for the dbcc page command is as follows: dbcc page (dbid | page_no [, 0 | 1 | 2]) The SQL Server keeps track of which object a page belongs to, if any. The allocation of pages within SQL Server is managed through the use of allocation units and allocation pages. Allocation Pages Space is allocated to a SQL Server database by the create database and alter database

commands. The space allocated to a database is divided into a number of 2KB pages.

\n

Each page is assigned a logical page number starting at page 0 and increased sequentially. The pages are then divided into allocation units of 256 contiguous 2KB pages, or 512 bytes (1/2 MB) each. The first page of each allocation unit is an allocation page that controls the allocation of all pages within the allocation unit. The allocation pages control the allocation of pages to tables and indexes within the database. Pages are allocated in contiguous blocks of eight pages called extents. The minimum unit of allocation within a database is an extent.

\n

When a table is created, it is initially assigned a single extent, or 16KB of space, even if the table contains no rows. There are 32 extents within an allocation unit (256/8). An allocation page contains 32 extent structures for each extent within that allocation unit. Each extent structure is 16 bytes and contains the following information: 1. Object ID of object to which extent is allocated 2. Next extent ID in chain 3. Previous extent ID in chain 4. Allocation bitmap 5. Deallocation bitmap 6. Index ID (if any) to which the extent is allocated 7. Status

\n

The allocation bitmap for each extent structure indicates which pages within the allocated extent are in use by the table. The deallocation bit map is used

to identify pages that have become empty during a transaction that has not yet been completed. The actual marking of the page as unused does not occur until the transaction is committed, to prevent another transaction from allocating the page before the transaction is complete. Data Pages A data page is the basic unit of storage within SQL Server. All the other types of pages within a database are essentially variations of the data page.

\n

All data pages contain a 32-byte header, as described earlier. With a 2KB page (2048 bytes) this leaves 2016 bytes for storing data within the data page. In SQL Server, data rows cannot cross page boundaries. The maximum size of a single row is 1962 bytes, including row overhead. Data pages are linked to one another by using the page pointers (prevpg, nextpg) contained in the page header. This page linkage enables SQL Server to locate all rows in a table by scanning all pages in the link. Data page linkage can be thought of as a two-way linked list.

\n

This enables SQL Server to easily link new pages into or unlink pages from the page linkage by adjusting the page pointers. In addition to the page header, each data page also contains data rows and a row offset table. The row-offset table grows backward from the end of the page and contains the location or each row on the data page. Each entry is 2 bytes wide. Data Rows Data is stored on data pages in data rows. The size of each data row is a factor of the sum of the size of the columns plus the row overhead. Each

record in a data page is assigned a row number. A single byte is used within each row to store the row number.

\n

Therefore, SQL Server has a maximum limit of 256 rows per page, because that is the largest value that can be stored in a single byte (2^8). For a data row containing all fixed-length columns, there are four bytes of overhead per row: 1. Byte to store the number of variable-length columns (in this case, 0) 1 byte to store the row number. 2. Bytes in the row offset table at the end of the page to store the location of the row on the page. If a data row contains variable-length columns, there is additional overhead per row. A data row is variable in size if any column is defined as varchar, varbinary, or allows null values.

\n

In addition to the 4 bytes of overhead described previously, the following bytes are required to store the actual row width and location of columns within the data row: 2 bytes to store the total row width 1 byte per variable-length column to store the starting location of the column within the row 1 byte for the column offset table 1 additional byte for each 256-byte boundary passed Within each row containing variable-length columns, SQL Server builds a column offset table backward for the end of the row for each variable-length column in the table.

\n

Because only 1 byte is used for each column with a maximum offset of 255, an adjust byte must be created for each 256-byte boundary crossed as an additional offset. Variable-length columns are always stored after all fixed-length columns, regardless of the order of the columns in the table definition. Estimating Row and Table Sizes Knowing the size of a data row and the corresponding overhead per row helps you determine the number of rows that can be stored per page.

\n

The number of rows per page affects the system performance. A greater number of rows per page can help query performance by reducing the number of ages that need to be read to satisfy the query. Conversely, fewer rows per page help improve performance for concurrent transactions by reducing the chances of two or more users accessing rows on the same page that may be locked. Let's take a look at how you can estimate row and table sizes. Fixed-length fields with no null values.

\n

Sum of column widths overhead- The Row Offset Table The location of a row within a page is determined by using the row offset table at the end of the page. To find a specific row within the page, SQL Server looks in the row offset table for the starting byte address within the data page for that row ID. Note that SQL Server keeps all free space at the end of the data page, shifting rows up to fill in where a previous row was deleted and ensuring no space fragmentation within the page.

\n

If the offset table contains a zero value for a row ID that indicates that the row has been deleted. Index Structure All SQL Server indexes are B-Trees. There is a single root page at the top of the tree, branching out into N number of pages at each intermediate level until it reaches the bottom, or leaf level, of the index. The index tree is traversed by following pointers from the upper-level pages down through the lower-level pages. In addition, each index level is a separate page chain. There may be many intermediate levels in an index.

\n

The number of levels is dependent on the index key width, the type of index, and the number of rows and/or pages in the table. The number of levels is important in relation to index performance. Non-clustered Indexes A non-clustered index is analogous to an index in a textbook. The data is stored in one place, the index in another, with pointers to the storage location of the data. The items in the index are stored in the order of the index key values, but the information in the table is stored in a different order (which can be dictated by a clustered index).

\n

If no clustered index is created on the table, the rows are not guaranteed to be in any particular order. Similar to the way you use an index in a book, Microsoft® SQL Server™ 2000 searches for a data value by searching the non-clustered index to find the location of the data value in the table and

then retrieves the data directly from that location. This makes non-clustered indexes the optimal choice for exact match queries because the index contains entries describing the exact location in the table of the data values being searched for in the queries.

\n

If the underlying table is sorted using a clustered index, the location is the clustering key value; otherwise, the location is the row ID (RID) comprised of the file number, page number, and slot number of the row. For example, to search for an employee ID (emp_id) in a table that has a non-clustered index on the emp_id column, SQL Server looks through the index to find an entry that lists the exact page and row in the table where the matching emp_id can be found, and then goes directly to that page and row. Clustered Indexes

\n

A clustered index determines the physical order of data in a table. A clustered index is analogous to a telephone directory, which arranges data by last name. Because the clustered index dictates the physical storage order of the data in the table, a table can contain only one clustered index. However, the index can comprise multiple columns (a composite index), like the way a telephone directory is organized by last name and first name. Clustered Indexes are very similar to Oracle's IOT's (Index-Organized Tables).

\n

A clustered index is particularly efficient on columns that are often searched for ranges of values. After the row with the first value is found using the

clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. For example, if an application frequently executes a query to retrieve records between a range of dates, a clustered index can quickly locate the row containing the beginning date, and then retrieve all adjacent rows in the table until the last date is reached. This can help increase the performance of this type of query.

\n

Also, if there is a column(s) that is used frequently to sort the data retrieved from a table, it can be advantageous to cluster (physically sort) the table on that column(s) to save the cost of a sort each time the column(s) is queried. Clustered indexes are also efficient for finding a specific row when the indexed value is unique. For example, the fastest way to find a particular employee using the unique employee ID column emp_id is to create a clustered index or PRIMARY KEY constraint on the emp_id column.

\n

Note PRIMARY KEY constraints create clustered indexes automatically if no clustered index already exists on the table and a non-clustered index is not specified when you create the PRIMARY KEY constraint. Index Structures Indexes are created on columns in tables or views. The index provides a fast way to look up data based on the values within those columns. For example, if you create an index on the primary key and then search for a row of data based on one of the primary key values, SQL Server first finds that value in the index, and then uses the index to quickly locate the entire row of data.

\n

Without the index, a table scan would have to be performed in order to locate the row, which can have a significant effect on performance. You can create indexes on most columns in a table or a view. The exceptions are primarily those columns configured with large object (LOB) data types, such as image, text, and varchar(max). You can also create indexes on XML columns, but those indexes are slightly different from the basic index and are beyond the scope of this article. Instead, I'll focus on those indexes that are implemented most commonly in a SQL Server database.

\n

An index is made up of a set of pages (index nodes) that are organized in a B-tree structure. This structure is hierarchical in nature, with the root node at the top of the hierarchy and the leaf nodes at the bottom, as shown in Figure 1. Figure 1: B-tree structure of a SQL Server index When a query is issued against an indexed column, the query engine starts at the root node and navigates down through the intermediate nodes, with each layer of the intermediate level more granular than the one above. The query engine continues down through the index nodes until it reaches the leaf node.

\n

For example, if you’re searching for the value 123 in an indexed column, the query engine would first look in the root level to determine which page to reference in the top intermediate level. In this example, the first page points the values 1-100, and the second page, the values 101-200, so the query

engine would go to the second page on that level. The query engine would then determine that it must go to the third page at the next intermediate level. From there, the query engine would navigate to the leaf node for value 123.

\n

The leaf node will contain either the entire row of data or a pointer to that row, depending on whether the index is clustered or nonclustered. Clustered Indexes A clustered index stores the actual data rows at the leaf level of the index. Returning to the example above, that would mean that the entire row of data associated with the primary key value of 123 would be stored in that leaf node. An important characteristic of the clustered index is that the indexed values are sorted in either ascending or descending order.

\n

As a result, there can be only one clustered index on a table or view. In addition, data in a table is sorted only if a clustered index has been defined on a table. Note: A table that has a clustered index is referred to as a clustered table. A table that has no clustered index is referred to as a heap. Nonclustered Indexes Unlike a clustered indexed, the leaf nodes of a nonclustered index contain only the values from the indexed columns and row locators that point to the actual data rows, rather than contain the data rows themselves.

\n

This means that the query engine must take an additional step in order to locate the actual data. A row locator’s structure depends on whether it points to a clustered table or to a heap. If referencing a clustered table, the row locator points to the clustered index, using the value from the clustered index to navigate to the correct data row. If referencing a heap, the row locator points to the actual data row. Nonclustered indexes cannot be sorted like clustered indexes; however, you can create more than one nonclustered index per table or view.

\n

SQL Server 2005 supports up to 249 nonclustered indexes, and SQL Server 2008 support up to 999. This certainly doesn’t mean you should create that many indexes. Indexes can both help and hinder performance, as I explain later in the article. In addition to being able to create multiple nonclustered indexes on a table or view, you can also add included columns to your index. This means that you can store at the leaf level not only the values from the indexed column, but also the values from non-indexed columns. This strategy allows you to get around some of the limitations on indexes.

\n

For example, you can include non-indexed columns in order to exceed the size limit of indexed columns (900 bytes in most cases). Index Types In addition to an index being clustered or nonclustered, it can be configured in other ways: * Composite index: An index that contains more than one column. In both SQL Server 2005 and 2008, you can include up to 16

columns in an index, as long as the index doesn’t exceed the 900-byte limit. Both clustered and nonclustered indexes can be composite indexes. * Unique Index: An index that ensures the uniqueness of each value in the indexed column.

\n

If the index is a composite, the uniqueness is enforced across the columns as a whole, not on the individual columns. For example, if you were to create an index on the FirstName and LastName columns in a table, the names together must be unique, but the individual names can be duplicated. A unique index is automatically created when you define a primary key or unique constraint: * Primary key: When you define a primary key constraint on one or more columns, SQL Server automatically creates a unique, clustered index if a clustered index does not already exist on the table or view.

\n

However, you can override the default behavior and define a unique, nonclustered index on the primary key. * Unique: When you define a unique constraint, SQL Server automatically creates a unique, nonclustered index. You can specify that a unique clustered index be created if a clustered index does not already exist on the table. * Covering index: A type of index that includes all the columns that are needed to process a particular query. For example, your query might retrieve the FirstName and LastName columns from a table, based on a value in the ContactID column.

\n

You can create a covering index that includes all three columns. Teradata What is the Teradata RDBMS? The Teradata RDBMS is a complete relational database management system. With the Teradata RDBMS, you can access, store, and operate on data using Teradata Structured Query Language (Teradata SQL). It is broadly compatible with IBM and ANSI SQL. Users of the client system send requests to the Teradata RDBMS through the Teradata Director Program (TDP) using the Call-Level Interface (CLI) program (Version 2) or via Open Database Connectivity (ODBC) using the Teradata ODBC Driver.

\n

As data requirements grow increasingly complex, so does the need for a faster, simpler way to manage data warehouse. That combination of unmatched performance and efficient management is built into the foundation of the Teradata Database. The Teradata Database is continuously being enhanced with new features and functionality that automatically distribute data and balance mixed workloads even in the most complex environments.

\n

Teradata Database 14 currently offers low total cost of ownership in a simple, scalable, parallel and self-managing solution. This proven, high-performance decision support engine running on the Teradata Purpose-Built PlatformFamilyoffers a full suite of data access and management tools, plus

world-class services. The Teradata Database supports installations from fewer than 10 gigabytes to huge warehouses with hundreds of terabytes and thousands of customers. Features & Benefits

\n

Automatic Built-In Functionality | Fast Query Performance | “ Parallel Everything” design and smart Teradata Optimizer enables fast query execution across platforms| | Quick Time to Value | Simple set up steps with automatic “ hands off” distribution of data, along with integrated load utilities result in rapid installations| | Simple to Manage | DBAs never have to set parameters, manage table space, or reorganize data| | Responsive to Business Change | Fully parallel MPP “ shared nothing” architecture scales linearly across data, users, and applications providing consistent and predictable performance and growth| Easy Set & G0” Optimization Options | Powerful, Embedded Analytics | In-database data mining, virtual OLAP/cubes, geospatial and temporal analytics, custom and embedded services in an extensible open parallel framework drive efficient and differentiated business insight| | Advanced Workload Management | Workload management options by user, application, time of day and CPU exceptions| | Intelligent Scan Elimination | “ Set and Go” options reduce full file scanning (Primary, Secondary, Multi-level Partitioned Primary, Aggregate Join Index, Sync Scan)| Physical Storage Structure of Teradata Teradata offers a true hybrid row and Column database.

\n

All database management systems constantly tinker with the internal structure of the files on disk. Each release brings an improvement or two that has been steadily improving analytic workload performance. However, few of the key players in relational database management systems (RDBMS) have altered the fundamental structure of having all of the columns of the table stored consecutively on disk for each record. The innovations and practical use cases of " columnar databases" have come from the independent vendor world, where it has proven to be quite effective in the performance of an increasingly important class of analytic query.

\n

These columnar databases store data by columns instead of rows. This means that all values of a single column are stored consecutively on disk. The columns are tied together as " rows" only in a catalog reference. This gives a much finer grain of control to the RDBMS data manager. It can access only the columns required for the query as opposed to being forced to access all columns of the row. It's optimal for queries that need a small percentage of the columns in the tables they are in but suboptimal when you need most of the columns due to the overhead in attaching all of the columns together to form the result sets.

\n

Teradata 14 Hybrid Columnar The unique innovation by Teradata, in Teradata 14, is to add columnar structure to a table, effectively mixing row structure, column structures and multi-column structures directly in the

DBMS which already powers many of the largest data warehouses in the world. With intelligent exploitation of Teradata Columnar in Teradata 14, there is no longer the need to go outside the data warehouse DBMS for the power of performance that columnar provides, and it is no longer necessary to sacrifice robustness and support in the DBMS that holds the post-operational data.

\n

A major component of that robustness is parallelism, a feature that has obviously fueled much of Teradata’sleadershipposition in large-scale enterprise data warehousing over the years. Teradata’s parallelism, working with the columnar elements, are creating an entirely new paradigm in analytic computing – the pinpoint accuracy of I/O with column and row partition elimination. With columnar and parallelism, the I/O executes very precisely on data interesting to the query. This is finally a strong, and appropriate, architectural response to the I/O bottleneck issue that analytic queries have been living with for a decade.

\n

It also may be Teradata Database’s most significant enhancement in that time. The physical structure of each container can also be in row (extensive page metadata including a map to offsets) which is referred to as “ row storage format,” or columnar (the row “ number” is implied by the value’s relative position). Partition Elimination and Columnar The idea of data division to create smaller units of work as well as to make those units of

work relevant to the query is nothing new to Teradata Database, and most DBMSs for that matter.

\n

While the concept is being applied now to the columns of a table, it has long been applied to its rows in the form of partitioning and parallelism. One of the hallmarks of Teradata’s unique approach is that all database functions (table scan, index scan, joins, sorts, insert, delete, update, load and all utilities) are done in parallel all of the time. There is no conditional parallelism. All units of parallelism participate in each database action. Teradata eliminates partitions from needing I/O by reading its metadata to understand the range of data placed into the partitions and eliminating those that are washed out by the predicates.

\n

See Figure There is no change to partition elimination in Teradata 14 except that the approach also works with columnar data, creating a combination row and column elimination possibility. In a partitioned, multi-container table, the unneeded containers will be virtually eliminated from consideration based on the selection and projection conditions of the query. See Figure Following the column elimination, unneeded partitions will be virtually eliminated from consideration based on the projection conditions.

\n

For the price of a few metadata reads to facilitate the eliminations, the I/O can now specifically retrieve a much focused set of data. The addition of

columnar elimination reduces the expensive I/O operation, and hence the query execution time, by orders of magnitude for column-selective queries. The combination of row and column elimination is a unique characteristic of Teradata’s implementation of columnar. Compression in Teradata Columnar Storage costs, while decreasing on a per-capita basis over time, are still consuming increasing budget due to the massive increase in the volume of data to store.

\n

While the data is required to be under management, it is equally required that the data be compressed. In addition to saving on storage costs, compression also greatly aids the I/O problem, effectively offering up more relevant information in each I/O. Columnar storage provides a unique opportunity to take advantage of a series of compression routines that make more sense when dealing with well-defined data that has limited variance like a column (versus a row with high variability.) Teradata Columnar utilizes several compression methods that take advantage of the olumnar orientation of the data. A few methods are highlighted below. Run-Length Encoding When there are repeating values (e. g. , many successive rows with the value of ‘ 12/25/11’ in the date container), these are easily compressed in columnar systems like Teradata Columnar, which uses “ run length encoding” to simply indicate the range of rows for which the value applies. Dictionary Encoding Even when the values are not repeating successively, as in the date example, if they are repeating in the container, there is

opportunity to do a dictionary representation of the data to further save space.

\n

Dictionary encoding is done in Teradata Columnar by storing compressed forms of the complete value. The dictionary representations are fixed length which allows the data pages to remain void of internal maps to where records begin. The records begin at fixed offsets from the beginning of the container and no “ value-level” metadata is required. This small fact saves calculations at run-time for page navigation, another benefit of columnar. For example, 1= Texas, 2= Georgia and 3= Florida could be in the dictionary, and when those are the column values, the 1, 2 and 3 are used in lieu of Texas, Georgia and Florida.

\n

If there are 1, 000, 000 customers with only 50 possible values for state, the entire vector could be stored with 1, 000, 000 bytes (one byte minimum per value). In addition to dictionary compression, including the “ trimming” 8 of character fields, traditional compression (with algorithm UTF8) is made available to Teradata Columnar data. Delta Compression Fields in a tight range of values can also benefit from only storing the offset (“ delta”) from a set value. Teradata Columnar calculates an average for a container and can store only the offsets from that value in place of the field.

\n

Whereas the value itself might be an integer, the offsets can be small integers, which double the space utilization. Compression methods like this lose their effectiveness when a variety of field types, such as found in a typical row, need to be stored consecutively. The compression methods are applied automatically (if desired) to each container, and can vary across all the columns of a table or even from container to container within a column9 based on the characteristics of the data in the container.

\n

Multiple methods can be used with each column, which is a strong feature of Teradata Columnar. The compounding effect of the compression in columnar databases is a tremendous improvement over the standard compression that would be available for a strict row-based DBMS. Teradata Indexes Teradata provides several indexing options for optimizing the performance of your relational databases. i. Primary Indexes ii. Secondary Indexes iii. Join Indexes iv. Hash Indexes v. Reference Indexes Primary Index Primary index determines the distribution of table rows on the disks controlled by AMPs.

\n

In Teradata RDBMS, a primary index is required for row distribution and storage. When a new row is inserted, its hash code is derived by applying a hashing algorithm to the value in the column(s) of the primary code (as show in the following figure). Rows having the same primary index value are stored on the same AMP. Rules for defining primary indexes The primary indexes for a table should represent the data values most used by the SQL to

access the data for the table. Careful selection of the primary index is one of the most important steps in creating a table.

\n

Defining primary indexes should follow the following rules: * A primary index should be defined to provide a nearly uniform distribution of rows among the AMPs, the more unique the index, the more even the distribution of rows and the better space utilization. * The index should be defined on as few columns as possible. * Primary index can be either Unique or non-unique. A unique index must have a unique value in the corresponding fields of every row; a non-unique index permits the insertion of duplicate field values. The unique primary index is more efficient. Once created, the primary index cannot be dropped or modified, the index must be changed by recreating the table. If a primary index is not defined in the CREATE TABLE statement through an explicit declaration of a PRIMARY INDEX, the default is to use one of the following: * PRIMARY key * First UNIQUE constraint * First column The primary index values are stored in an integral part of the primary table. It should be based on the set selection most frequently used to access rows from a table and on the uniqueness of the value.

\n

Secondary Index In addition to a primary index, up to 32 unique and non-unique secondary indexes can be defined for a table. Comparing to primary indexes, Secondary indexes allow access to information in a table by alternate, less frequently used paths. A secondary index is a subtable that is

stored in all AMPs, but separately from the primary table. The subtables, which are built and maintained by the system, contain the following; * RowIDs of the subtable rows * Base table index column values * RowIDs of the base table rows (points)

\n

As shown in the following figure, the secondary index subtable on each AMP is associated with the base table by the rowID . Defining and creating secondary index Secondary index are optional. Unlike the primary index, a secondary index can be added or dropped without recreating the table. There can be one or more secondary indexes in the CREATE TABLE statement, or add them to an existing table using the CREATE INDEX statement or ALTER TABLE statement. DROP INDEX can be used to dropping a named or unnamed secondary index.

\n

Since secondary indexes require subtables, these subtables require additional disk space and, therefore, may require additional I/Os for INSERTs, DELETEs, and UPDATEs. Generally, secondary index are defined on column values frequently used in WHERE constraints. Join Index A join index is an indexing structure containing columns from multiple tables, specifically the resulting columns form one or more tables. Rather than having to join individual tables each time the join operation is needed, the query can be resolved via a join index and, in most cases, dramatically improve performance.

\n

Effects of Join index Depending on the complexity of the joins, the Join Index helps improve the performance of certain types of work. The following need to be considered when manipulating join indexes: * Load Utilities The join indexes are not supported by MultiLoad and FastLoad utilities, they must be dropped and recreated after the table has been loaded. * Archive and Restore Archive and Restore cannot be used on join index itself. During a restore of a base table or database, the join index is marked as invalid.

\n

The join index must be dropped and recreated before it can be used again in the execution of queries. * Fallback Protection Join index subtables cannot be Fallback-protected. * Permanent Journal Recovery The join index is not automatically rebuilt during the recovery process. Instead, the join index is marked as invalid and the join index must be dropped and recreated before it can be used again in the execution of queries. * Triggers A join index cannot be defined on a table with triggers. Collecting Statistics In general, there is no benefit in collecting statistics on a join index for joining columns specified in the join index definition itself. Statistics related to these columns should be collected on the underlying base table rather than on the join index. Defining and creating secondary index Join indexes can be created and dropped by using CREATE JOIN INDEX and DROP JOIN INDEX statements. Join indexes are automatically maintained by the system when updates (UPDATE, DELETE, and INSERT) are performed on the underlying base tables.

\n

Additional steps are included in the execution plan to regenerate the affected portion of the stored join result. Hash Indexes Hash indexes are used for the same purposes as single-table join indexes. The principal difference between hash and single-table join indexes are listed in the following table. Hash indexes create a full or partial replication of a base table with a primary index on a foreign key column table to facilitate joins of very large tables by hashing them to the same AMP. You can define a hash index on one table only.

\n

The functionality of hash indexes is a superset to that of single-table join indexes. Hash indexes are not indexes in the usual sense of the word. They are base tables that cannot be accessed directly by a query. The Optimizer includes a hash index in a query plan in the following situations. * The index covers all or part of a join query, thus eliminating the need to redistribute rows to make the join. In the case of partial query covers, the Optimizer uses certain implicitly defined elements in the hash index to join it with its underlying base table to pick up the base table columns necessary to complete the cover. A query requests that one or more columns be aggregated, thus eliminating the need to perform the aggregate computation For the most part, hash index storage is identical to standard base table storage except that hash indexes can be compressed. Hash index rows are hashed and partitioned on their primary index (which is always defined as non-unique). Hash index tables can be indexed explicitly, and

their indexes are stored just like non-unique primary indexes for any other base table.

\n

Unlike join indexes, hash index definitions do not permit you to specify secondary indexes. The major difference in storage between hash indexes and standard base tables is the manner in which the repeated field values of a hash index are stored. Reference Indexes A reference index is an internal structure that the system creates whenever a referential integrity constraint is defined between tables using a PRIMARY KEY or UNIQUE constraint on the parent table in the relationship and a REFERENCES constraint on a foreign key in the child table.

\n

The index row contains a count of the number of references in the child, or foreign key, table to the PRIMARY KEY or UNIQUE constraint in the parent table. Apart from capacity planning issues, reference indexes have no user visibility.

\n\n

References for Teradata

\n

\n \t

1. http://www. teradata. com/products-and-services/database/ \n \t
2. http://teradata. uark. edu/research/wang/indexes. html \n \t

3. http://www. teradata. com/products-and-services/database/teradata-13/ \n \t
4. http://www. odbms. org/download/illuminate%20Comparison. pdf \n