IBM Db2 for i indexing methods and strategies

Learn how to use Db2 indexes to boost performance

Kent Milligan
Michael Cain
Db2 for i Team
IBM Technology Expert Labs

June 2023

Table of contents

Abstract

This white paper lays the foundation for an indexing strategy and design that delivers high-performance queries and SQL applications on IBM Db2 for i. Both, programmers and database engineers can find information on indexing to make their jobs easier and improve the performance of their Db2 for i servers. This in-depth discussion on Db2 for i indexing includes a description of the technology along with coverage of the Db2 performance tools available to assist with index analysis and SQL performance tuning.

For those new to the world of database indexing, the paper covers the basics of how indexes are used by the Db2 for i and best practices for creating the optimal set of indexes for the Db2 for i query optimizer. Experienced database users can be educated on the latest Db2 for i indexing technologies including the encoded vector index (EVI) support, index ANDing and index ORing, sparse indexes, and derived key indexes. All levels of readers can find a large number of indexing examples to assist them in building a deeper knowledge of Db2 for i indexing best practices.

Introduction

On any platform, good database performance depends on a proper architecture and good design. And, good design includes a solid understanding of indexes and statistics: how many to build, their structure and complexity, as well as their maintenance requirements.

The importance of indexes and statistics also holds true for IBM® Power® processor-based servers running the IBM i operating system and its integrated IBM Db2® for i database. This is the reason why Db2 for i provides a robust set of technologies for indexing and contains an advanced cost-based query optimizer that understands how to best utilize these index technologies. Using Db2 for i indexing support is a powerful tool, but also requires knowledge and planning regarding their creation and application. Having the right set of indexes available for the Db2 for i query optimizer to use is a critical success factor in delivering high-performing SQL applications and reporting workloads.

This paper provides an initial look at indexing strategies and their effects on SQL and query performance. Thus, it is strongly recommended that database administrators, engineers, analysts, and developers who are new to Db2 for i or using SQL on IBM i, learn the skills covered in the Database Engineer (DBE) Information offering. This offering provides in-depth information on the way to architect and implement a high-performing and scalable Db2 for i solution. You can find more information about this offering: ibm.biz/Db2iExpertLabs

As the IBM i predecessor platforms, IBM System/38 and IBM AS/400, were designed before SQL was widely used, IBM had to create a non-SQL, a proprietary interface for relational database creation and data access. This propriety interface, also known as the native database interface, is comprised of data description specifications (DDS) for creating Db2 objects and the record-level access APIs for data processing and retrieval. Because of this native interface and history, some IBM i users use a terminology that is not familiar to those coming from an SQL background when discussing database indexing strategies. Here is a table that maps the SQL terminology with the native IBM i terminology:

SQL term	**IBM i term**
TABLE	PHYSICAL FILE
ROW	RECORD
COLUMN	FIELD
INDEX	KEYED LOGICAL FILE
VIEW	NON-KEYED LOGICAL FILE
SCHEMA	LIBRARY
LOG	JOURNAL
ISOLATION LEVEL	COMMITMENT CONTROL LEVEL
PARTITION	MEMBER

Table 1: Mapping SQL terminology with IBM i terminology

The Db2 for i database was actually one of the first databases to achieve compliance with the core level of the SQL 2003 and 2008 standards. Due to the integrated nature of the IBM i relational database management system, the use of both its native and SQL interfaces are almost completely interchangeable. Objects created with DDS can be accessed with SQL statements; and objects created with SQL can be accessed with the native record-level access APIs, assuming that the native record level access interface supports the SQL data types involved.

The IBM i native interfaces are widely used and continue to be supported by IBM. However, these native interfaces are not being enhanced with the same functionality as the strategic Db2 for i interface – SQL. As a result, IBM i developers need to start adopting and using SQL. For a better understanding of the benefits of using SQL, refer to the *DDS and SQL - A Winning Combination for Db2 for i* white paper at: **ibm.biz**/db2iPapers

The basics

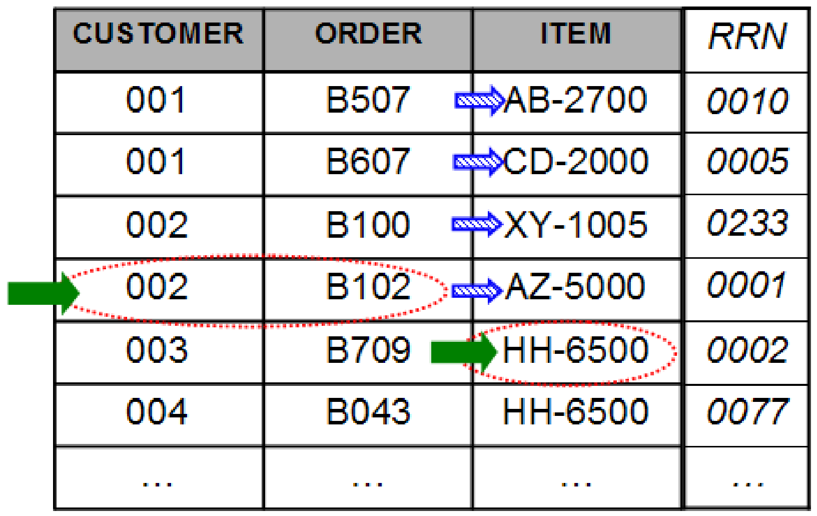
Before describing strategies and options for index creation and use, it is important to understand what an index is, the types of indexes that are available, and how indexes are used by the Db2 for i engine.

Database index introduction

The purpose of a book's index section is to give the reader a faster and more efficient way to locate information on a specific topic. A good index first organizes the book's topics into a structure that is useful and easily understood by the reader, and then provides the pages numbers for this information so that the reader can go directly to the pages of interest. Without an index, the reader is instead required to read or scan each page of the book to locate the same information.

Database indexes in relational database management systems (RDBMS) provide similar benefits by providing a relatively fast method of locating data of interest. Without indexes, the database will likely perform a full sequential search or scan of the table, accessing every row in the table. Depending on the size of the table and the complexity of the query, a full table scan can be a lengthy process and consume a large amount of system resources. One of the most common causes of poor SQL performance with any RDBMS is missing or suboptimal indexes.

Although index structures vary from product to product, most RDBMS products speed up SQL performance by using an index in one of two ways – an index probe or index scan. These two index operations are graphically represented in Figure 1. Both, the index probe and scan operations, locate the row in the table to process by first searching for the specified key value and then using the relative row number (RRN) or row identifier (RID) stored alongside the matching key value. For example, customer 003 in Figure 1 would be found in row number 2 of the underlying table.



CUSTOMER	ORDER	ITEM	*RRN*
001	B507	AB-2700	*0010*
001	B607	CD-2000	*0005*
002	B100	XY-1005	*0233*
002	B102	AZ-5000	*0001*
003	B709	HH-6500	*0002*
004	B043	HH-6500	*0077*
…	…	…	…

Index Key Columns (CUSTOMER, ORDER, ITEM)

Figure 1 - Index probe and scan comparison

An index probe is the most efficient method because most database engines can directly position to the index key values specified for the search criteria (for example, WHERE order = 'B102' AND customer = '002'). An index probe operation can only be performed when the columns being searched matched the leading, contiguous key columns of an index. When the search columns do not match the leading key columns of an index, the index scan method can be utilized. The index scan operation has to compare (represented by shaded arrow) each key value in the index before locating the key value that meets the search criteria (for example, WHERE item = 'HH-6500').

When a small percentage of the rows are being retrieved, index probes and scans are typically more efficient than table scans since the index key values are usually shorter in length than the length of the database table row. Shorter entries means that more index entries can be stored on a single disk page. It is pretty common for an index page to contain at least ten times more key values than the number of rows stored on a single page in the table. Thus, a proper indexing scheme results in a considerable reduction in the total number of pages that must be processed with disk I/O requests in order to locate the requested data.

While indexes can improve performance, the complexity of the query and the data will determine how effectively the data access can be implemented. Different queries stress the database in unique ways and that is why different types of indexes are needed to cope with ever-changing requests by users. In addition to simply retrieving data more efficiently, indexes can also assist in the ordering, grouping, and joining of data from different tables. The Db2 for i query optimizer also relies on indexes to provide statistics - more on statistics later.

Db2 for i indexing technology

With Db2 for i, there are two kinds of relational index technologies: radix indexes, which have been available since IBM AS/400 systems began shipping in 1988, and encoded vector indexes (EVIs), which became available in 1998 with IBM OS/400 Version 4 Release 3. Both types of indexes are useful in improving performance for certain types of queries as well as providing statistics to the Db2 for i query optimizer to enhance its decision-making process.

A new type of index was introduced with the no-charge IBM OmniFind® Text Search Server product (5733-OMF). The IBM OmniFind Text Search Server enables applications to perform advanced, high-speed linguistic searches against both simple character data and rich-text documents (Adobe PDF, Microsoft® Word, and so on). However, usage of the IBM OmniFind text search indexes is limited to the SQL Contains and Score functions.

Radix indexes

All commercially available RDBMS use some form of binary tree index structure. As Figure 2 demonstrates, a radix index is a multilevel, hybrid tree structure that allows a large number of key values to be stored efficiently, while minimizing access times. A key compression algorithm assists in this process. The Db2 for i radix index object also contains metadata and statistics about the actual key values (such as number of keys, key cardinality for single and composite keys, and so on.).

The lowest level of the index tree structure contains the leaf nodes that house the address of the rows in the base table that are associated with the key value. The key value is used to quickly navigate to the leaf node with a few simple binary search tests.

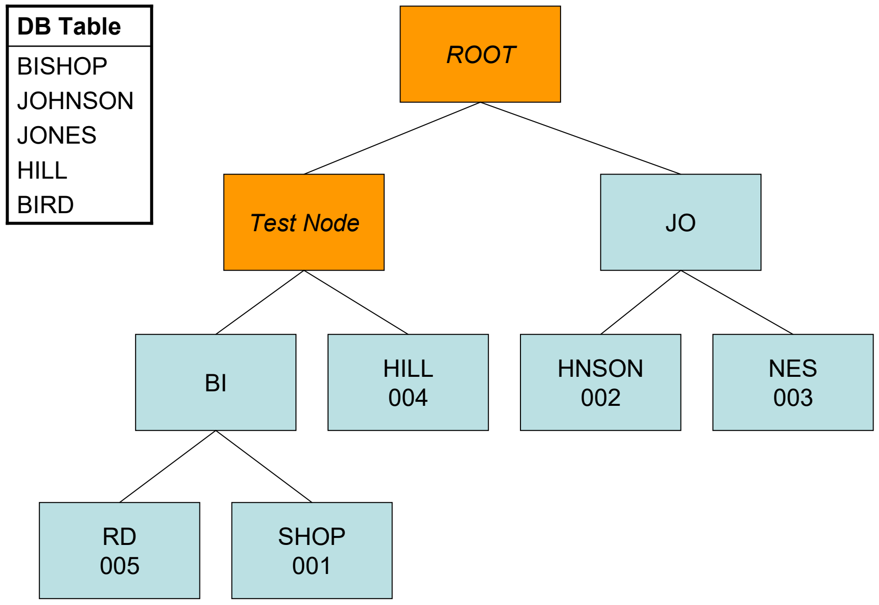


Figure 2 – Example of a radix index structure

Thus, a single key value can be accessed quickly with a small number of tests. This quick access is very consistent across all key values in the index as the operating system automatically keeps the depth of the index shallow and the index pages spread across multiple disk units.

The radix tree structure is the default structure used by Db2 for i when indexes are created using either the SQL **Create Index** statement or the **Create Logical File** (CRTLF) system command. In addition, Db2 for i always uses a radix index to enforce primary key, unique key, or foreign key constraints whether it has to use an existing index or create a new index.

The radix tree structure is very good for finding one row or a relatively small number of rows because it is able to identify a given row with a minimal amount of processing effort. For example, using a radix index over a customer number column for a typical online transaction processing (OLTP) request, such as *find the outstanding orders for a single customer*, results in a very fast retrieval performance. An index created over the customer number column would be considered the perfect index for this type of query because it allows the database to focus on only the rows it needs, while performing a minimal number of I/Os.

The downside of a radix index is that the table rows are retrieved in order of key value instead of physical order, which results in random I/O requests against the table instead of sequential I/O processing. For example, the first key value is **BIRD** which is found in the fifth row of the table while the second key value **BISHOP** is found in the first row of the table. The random I/O nature can cause performance inefficiencies when a large number of key values are retrieved. When traversing a radix index for a large number of keys, there is also no way to predict the physical index pages that need to be accessed next.

In business intelligence environments, the queries that analysts are trying to optimize are not as predictable as they are for OLTP environments. Increasingly, users require the ability to submit ad-hoc queries against the detail data underlying their data marts. They might, for example, run a report every week to look at sales data, then drill-down for more information related to a particular problem area they found in the report. In this scenario, the database analysts cannot write all the queries in advance for the user. Without knowing what queries will be run ahead of time, it is impossible to build the perfect index.

Traditionally, the solution to this dilemma has been to either restrict ad-hoc query capability or define a set of indexes that cover most columns for most queries. With Db2 for i, the query optimizer can intelligently use less-than-perfect radix indexes for many types of queries. However, as the size of data warehouses grows into the terabyte range, less than perfect becomes less palatable. Thus, there is a need for an alternative indexing technology on IBM i.

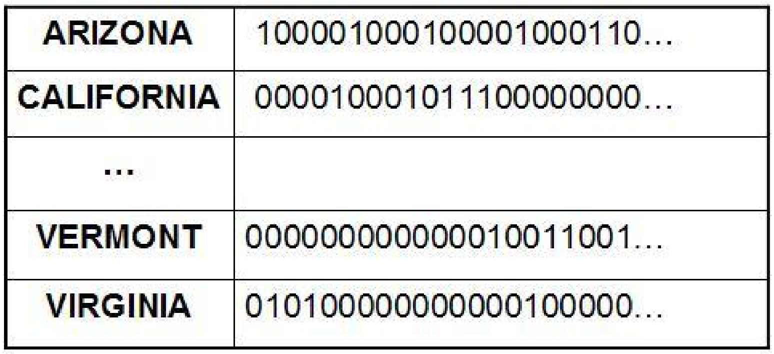
Encoded vector indexes

Experts throughout the industry recognized the limitations of indexes based on the binary tree structure and developed new types of indexes that can be combined dynamically at runtime to cover a broader range of ad-hoc queries. Bitmap index technology emerged as one of the solutions to this problem.

However, the IBM Db2 for i development team chose not to support bitmap indexes because of limitations associated with that technology. Instead, Db2 for i includes support for the encoded-vector index technology. The encoded-vector index support is based on patented technology from IBM Research that addresses the limitations, while still providing a performance boost for ad-hoc queries. Before going into the details of encoded vector index technology, it is first necessary to understand the limitations of the bitmap index technology.

Bitmap indexes - the limitations

The concept of a bitmap index is an array of distinct key values. For each value, the index stores a bitmap, where each bit represents a row in the table containing that value, as shown in Figure 3. If the bit is set on (1), then that row contains the specific key value. If the bit is set off (0), then that row does not contain the specific key value. This support enables an RDBMS to quickly find all of the rows that contain a specific key value.



ARIZONA	100001000100001000110…
CALIFORNIA	000010001011100000000…
…	
VERMONT	000000000000010011001…
VIRGINIA	010100000000000100000…

Figure 3 – Example of a bitmap index structure

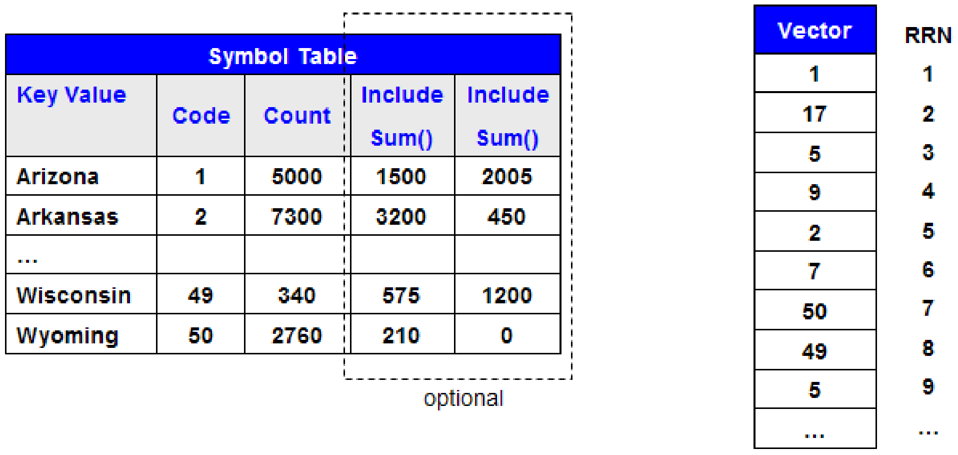
With this indexing scheme, bitmaps can be combined dynamically – at run time using Boolean arithmetic to identify only those rows that are required by the query. Unfortunately, this improved access comes with a price. In a very large database environment, bitmap indexes can grow to unmanageable sizes. In a one billion row table, for example, there will be one billion bits for each distinct key value. If the table contains many distinct values, the bitmap index quickly becomes enormous. Usually, the RDBMS relies on some sort of compression algorithm to help alleviate this growth problem.

In addition, maintenance of very large bitmap indexes can be problematic. Every time the database is updated, the system must update each bitmap — a potentially tedious process if there are, say, thousands of unique values in a large table. When adding a new distinct key value, an entire bitmap must be generated. These issues typically result in the database being used in a read-only manner.

EVI structure details

Figure 4 demonstrates the internal makeup of the EVI that is created by the SQL statement, **CREATE ENCODED VECTOR INDEX myevi ON sales(state)**. The EVI structure can only be utilized with SQL.

The EVI data structure is comprised of two basic components: the symbol table and the vector. The symbol table contains a distinct key list, along with statistical and descriptive information about each distinct key value in the index. The symbol table maps each distinct value to a unique code. The mapping of any distinct key value to a 1-, 2-, or 4-byte code provides a type of key compression. A key value, of any length, can be represented by a 1-, 2-, or 4-byte code. The optimizer can use the symbol table to obtain statistical information about the data and key values represented in the EVI.



Symbol Table				
Key Value	Code	Count	Include Sum()	Include Sum()
Arizona	1	5000	1500	2005
Arkansas	2	7300	3200	450
...				
Wisconsin	49	340	575	1200
Wyoming	50	2760	210	0

Vector	RRN
1	1
17	2
5	3
9	4
2	5
7	6
50	7
49	8
5	9
...	...

Figure 4 - Structure of an EVI

The other component, the vector, contains a byte code value for each row in the table. This byte code represents the actual key value found in the symbol table and the respective row in the database table. The byte codes are in the same ordinal position in the vector as the row it represents in the table. The vector does not contain any pointer or explicit references to the data in the underlying table. The single vector is a key reason that an EVI has lower maintenance costs than a traditional bitmap index. When a key value changes, there is only vector (that is, array) that needs to be updated instead of multiple bitmap arrays.

EVI runtime usage

When the Db2 optimizer decides to use an EVI to process the local selection of the query, the database engine uses the vector to build a dynamic bitmap, which contains one bit for each row in the table. As you might expect, this dynamic bitmap delivers the same runtime performance improvement as a bitmap index. Figure 5 shows how each bit in the bitmap corresponds to the same ordinal position as the row it represents in the underlying table. If the row satisfies the query, the bit is set on. If the row does not satisfy the query, the bit is set off. The database engine can also build a list of relative row numbers (RRNs) from the EVI. These RRNs represent the rows that match the selection criteria, without the need for a bitmap. The RRNs are naturally ordered as they are produced. In other words, the RRNs are sorted as a natural byproduct of scanning the vector. This phenomenon will be especially important during the reading of rows in the table. The rows that are of interest will be visited in their physical order allowing for much smoother and more efficient processing (for example, sequential access compared with random access). This processing is sometimes referred to as skip sequential processing because Db2 is able to skip over ranges of rows that do not meet the selection criteria. Furthermore, this positive behavior is available from any EVI, not just a single so called clustered index found in other database management systems.

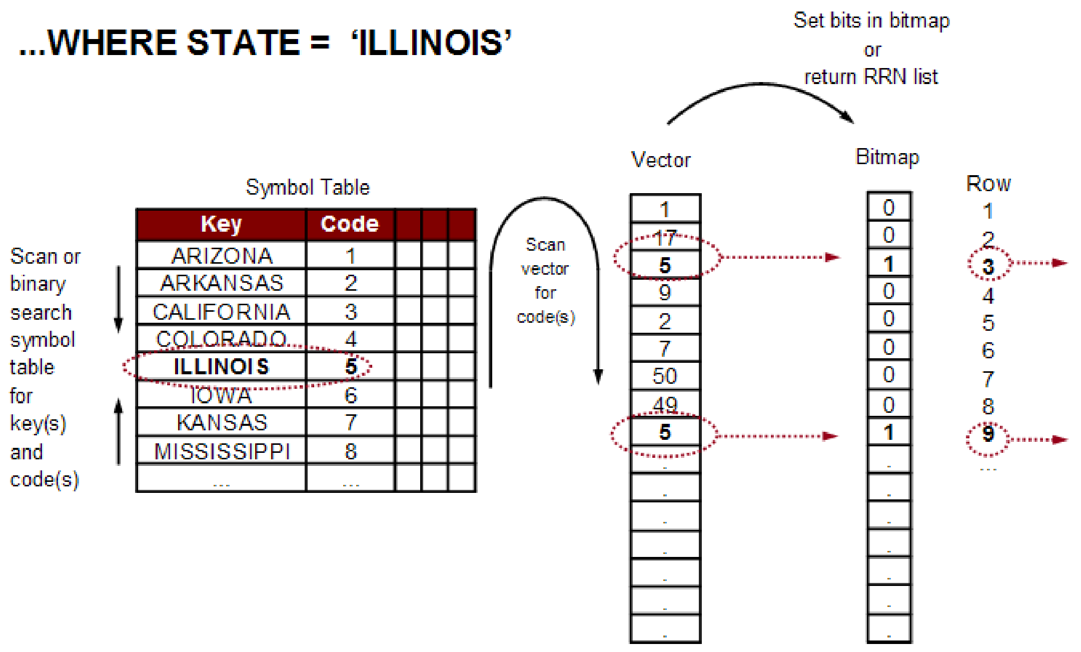


Figure 5 - EVI runtime dynamic bitmap

As with traditional bitmap indexes, the Db2 produced dynamic bitmaps or RRN lists can be merged together with Boolean arithmetic using logical AND operators or logical OR operators to satisfy an ad-hoc query. For example, if a user wants to view sales data for a certain region during a specific time period, the database analyst can define an EVI over the Region column and an EVI over the **Quarter** column of the database. When the query runs, the database engine builds an RRN list (or bitmap) using the two EVIs and then merge the RRN lists together to produce a bitmap that represents all the local selection (RRN for only the relevant rows). This capability to merge together dynamic bitmaps or RRN lists with index ANDing and index ORing technology can drastically reduce the number of rows that the database engine must retrieve and process.

The database engine can use the encoded vector index for index-only access processing and thus avoid the need to read the table. For example, a query that requests a count of the number of customers in each state can be implemented by Db2 for i simply by processing the contents of the symbol table. Given that the EVI symbol table can be expanded to include aggregated column sum, an EVI becomes a very powerful query optimization strategy to reduce the number of database I/O requests and associated database processing.

- As just mentioned, the EVI symbol table can be utilized to include and maintain aggregated column sums. The INCLUDE clause is used in the following **Create Index** statement in Listing 1 to add the summary of sales total for each state key value in the symbol table. The INCLUDE clause can be used with the following built-in aggregation functions: AVG, COUNT, COUNT_BIG, SUM, STDDEV, STDDEV_SAMP, VARIANCE, and VARIANCE_SAMP

```
    CREATE ENCODED VECTOR INDEX myevi2
          ON sales(state) INCLUDE ( SUM(saleamt) )
          WITH 50 DISTINCT VALUES
```

Listing 1 – EVI Include aggregate example

When an SQL statement requests the SUM of the saleamt column for each state using a GROUP BY clause that references the state column (that is, SELECT state, SUM(saleamt) FROM sales GROUP BY state), Db2 for i can very quickly return the aggregated sales amount total for each state by performing index-only access on the symbol table. All access and processing of the underlying table rows can be eliminated.

As EVIs were created primarily to support business intelligence and ad-hoc query environments, there are EVI creation and maintenance considerations, as well as recommendations — both of which are covered in later sections.

Derived and sparse indexes

Db2 for i also provides the ability to create derived key indexes (also known as function-based indexes) and sparse indexes. Both radix and encoded vector indexes can include key derivations and selection criteria in the index definition.

The derived key index support is a significant addition because on previous releases, the usage of a function on a search predicate prevented the optimizer from using an index to speed up the selection processing. The SELECT statements in Listing 2 are an example of the type of statements that were unable to benefit from the usage of an index in prior releases.

```
SELECT * FROM table1 WHERE UPPER(lastname) = 'CAIN'
SELECT * FROM table1 WHERE YEAR(shipdate) = 2010
```

Listing 2 - Example SQL statements with function-based search predicates

Now with the derived key index support, the indexes in Listing 3 can be created to give the query optimizer a way to speed up query performance. The ability to create an index key from the result of the UPPER function offers huge performance potential because many applications wrap the UPPER function around character columns as a way to implement case-insensitive searches.

```
CREATE INDEX upper_lastname_ix ON table1( UPPER(lastname) )
CREATE INDEX shipdate_year_ix ON table1( YEAR(shipdate) )
```

Listing 3 - SQL derived index examples

Adding a WHERE clause to the **Create Index** statement creates what is known as a sparse index. A normal index contains a key value for every row in the table while a sparse index only contain a key value for a subset of the rows in the table. The condition on the WHERE clause effectively limits the key values to only those rows in the table that meet the specified search condition or conditions. Listing 4 contains two example sparse indexes. The first index, blue_index, only contains key values for those rows in the items table that are blue in color. On the second index definition, notice that the WHERE clause references a column (activeCust) that is not part of the key definition (cust_id). As a result, the activeCust_index will only contain customerID values for those customers that have been flagged as active with the activeCust column.

```
CREATE INDEX blue_index ON items(color) WHERE color='BLUE'
CREATE INDEX activeCust_index ON customers(customerID) WHERE activeCust='Y'
```

Listing 4 - Sparse index definition examples

The queries found in Listing 5 are SQL requests that can potentially benefit from the optimizer utilizing the blue_index or activeCust_index. The Db2 for i support for sparse SQL indexes is only useful for those developers looking to replace the DDS definitions of select / omit logical files with the equivalent SQL **Create Index** statement.

```
SELECT COUNT(*) FROM items WHERE color='BLUE'
SELECT customer_address FROM customers WHERE activeCust='Y'
```

Listing 5 - SQL statements potentially benefiting from sparse indexes

Given the vast array of expressions and virtually unlimited combinations available in SQL, the query optimizer does have some restrictions in terms of the types of derived and sparse indexes that it is able to utilize in a query plan. Only the SQL Query Engine (SQE) optimizer has the ability to fully use derived and sparse indexes to speed up the performance of SQL requests. The Classic Query Engine (CQE) support for derived and sparse indexes is very limited. After creating a sparse or derived index, it is wise to check the usage of the index.

Db2 for i indexing comparison with other databases

In this section, Db2 for i index support is compared with the indexing technologies on other relational database management systems.

Besides the comparison points in this section, the Db2 for i indexing technology is an industry leader when it comes to the administration and management of the index objects. With Db2 for i, there is no requirement to periodically reorganize or rebalance the index structures. That is due to the fact that the database engine automatically performs these tasks as part of the normal index maintenance processing that occurs as key values are being inserted, updated, and deleted.

Primary and clustered indexes

On platforms that rely on partitioning schemes, the database must distinguish between primary and secondary indexes. Primary indexes are those created with the partitioning key as the primary key. Secondary indexes are built over columns other than the partitioning key. On these platforms, primary indexes provide the majority of data retrieval.

Because of the integrated storage management architecture of IBM i, the table objects are automatically spread across all available disk units while the difference between main storage (memory) and auxiliary storage (disk) is transparent to the user (that is, single level storage). One result is that all indexes are effectively primary indexes. In fact, there is no distinction between primary and secondary indexes within Db2 for i.

There is also no concept of a clustered index — where the table data is kept in the same physical order as the primary index key(s). Nevertheless, the problems of random reads can be overcome by many clever Db2 for i query methods and strategies such as the usage of EVIs and sorted lists of relative row numbers produced by a radix indexes.

The net result of the integrated and automatic storage management system is that there is no need to consider the physical storage and placement of Db2 for i tables or indexes.

Partitioned and non-partitioned indexes

Db2 for i supports local table partitioning with Db2 Multisystem offering, which is a separately licensed feature of the IBM i operating system (option 27). This feature allows a table's rows to be physically separated into individual data spaces identified by a partition key. Both range and hash partitioning are supported. A partitioned table can be indexed with a non-partitioned or spanning index, or a partitioned index. A spanning index contains keys that reference all rows in all partitions. A partitioned index only references rows in a specific partition. More information on local table partitioning and indexing partition tables can be found in the paper titled: *Table partitioning strategies* - **ibm.biz**/db2iPapers

Bitmapped indexing

The comparison of Db2 for i indexing technology with bitmapped index support is found in the "Encoded vector indexes" section.

Db2 for i usage of indexes

Db2 for i relies on database indexes to provide runtime implementation methods and statistics about the data. Utilizing indexes at runtime to quickly locate date values or sort data is a common practice on all RDBMS products. Relying on indexes to provide statistical insights about the data (for example, the number of distinct values or the selectivity of a search argument) is a capability unique to Db2 for i.

During the query optimization process, Db2 examines all of the indexes that are a relevant to the query being optimized. An index is viewed as relevant only when it contains keys that can either provide statistics or provide a runtime implementation method for the query. Consider this Select statement, **SELECT col1 WHERE col2 > 100 ORDER BY col3**. In this case, an index with **col1** as the key would be deemed uninteresting from an optimization perspective because it cannot provide statistics or runtime methods. Furthermore, the index does not contain all the columns referenced by the query, so index-only access is not possible for this Select statement, the query optimizer would only process indexes that have **col2** or **col3** as the leading column in the key definition. Note that only the leading contiguous key columns can be probed.

Statistical usage

The Db2 for i optimizer uses the statistical information from indexes to better understand the data stored in the underlying tables. A deeper understanding of the data helps the query optimizer find the most-efficient data access method for a given query request. Both radix and encoded vector indexes contain information about the number of distinct values in a column and the distribution of values.

With radix indexes, the optimizer obtains information from the left-most, contiguous keys in the index. In addition to knowing how many distinct values are in each column, radix indexes provide cross-column cardinality information. That is, the optimizer can look at the left-most columns in an index and determine how many distinct permutations of the column values exist in table. To obtain this statistical information, the Db2 engine estimates the number of rows (keys) that match the selection criteria by sampling portions of the radix index tree. This estimate process is part of the query optimization and uses the index to count a subset of the keys — thus providing the optimizer with a good insight on the selectivity of a query.

In contrast, the statistics provided by encoded vector indexes all come from the EVI symbol table object. All of the statistics provided by EVI are the actual live counts as opposed to the estimated statistics retrieved from radix indexes.

It is problematic for RDBMS to accurately represent the cardinality of columns. For example, an optimizer may assume that the distinct values are equally distributed throughout the table. For example, if a table contains a **State** column and the data reflects all 50 states in the United States, an optimizer might assume that each state appears equally (the data distribution for any state is 1/50th of the total rows). But as anyone familiar with the population distribution of the United States knows that it is very unlikely that the table will contain as many entries for North Dakota as it does for California. However, in Db2 for i, an index built over the State column will give the optimizer information about how many rows satisfy **North Dakota** and how many satisfy **California**. And if the distributions vary widely, the optimizer might build different access plans based on the actual **State** value specified in the query request. The number of distinct values in a key column or composite key can be accessed by using IBM i Access Client Solutions (ACS) to view the properties of an index.

SQE automatically gathers and maintains column statistics within the table object. The columns statistics provide information to the query optimizer when an index is not available to provide statistics. The automatic collection of column statistics happens in the background whenever the query optimizer encounters a column that is not the leading key column of an index.

Because the information that the optimizer needs is gathered automatically by the Db2 engine, and maintained in the Db2 objects themselves, administrators should never have to manually compile statistics for the optimizer.

More recently, Db2 for i has incorporated enhancements to automatically watch, learn, and capture statistics from the actual executions of queries. This capability is particularly important for complex predicates with derivations and expressions. For example, it is impossible to create an index or obtain a column statistic for a query expression such as (WHERE datecol = CURRENT_DATE - 30 DAYS + 1 YEAR). But with Db2 for i adaptive query processing (AQP), the system can gather and store the information derived from the real-time calculation. This information can be automatically used to enhance future query executions that contain the same expression.

Implementation usage

Most importantly, the database engine can use indexes to identify and process rows in a table. As in any RDBMS, the query optimizer can choose whether or not to use an index. In general, the optimizer chooses the index that will efficiently narrow down the number of rows matching the query selection, as well as for joining, grouping, and ordering operations. Put another way, the index is used to logically organize and order rows by a given key, and to reduce the number of I/Os required to retrieve the data needed to complete the request.

In order to process a query, the database must build an access plan. Think of the access plan as a recipe, with a list of ingredients and methods for cooking the dish. The query optimizer is the component of Db2 that builds the recipe. The ingredients are the tables and indexes required for the query. The optimizer looks at the methods it has available for a given query, estimates which ones are the most cost-effective, and builds a set of instructions on how to use the methods and ingredients. The Db2 database engine uses these instructions to do the cooking and retrieve the specified result set.

As the Db2 for i query optimizer uses cost-based optimization, the more the information available about the rows and columns in the database, the better the ability of the optimizer at creating the best possible (least costly / fastest) access plan for the query. With the information from the indexes, the optimizer can make better choices about how to process the request (local selection, joins, grouping, and ordering).

The primary goal of the optimizer is to choose an implementation that quickly and efficiently eliminates the rows *that are not interesting* or not required to satisfy the request. In other words, query optimization is concerned with trying to identify the rows of interest while avoiding useless data. A proper indexing strategy assists the optimizer and database engine with this task.

To understand indexing strategy, it is important to understand the science of query optimization and the possible implementation methods. With the limited scope of this paper, the implementation methods are explained only at a high level, tying the use of indexes to the respective methods. Here is an overview of the available implementation methods:

Selection

- Table scan
- Table probe*
- Index scan* (also known as index selection)
- Index probe* (also known as key row positioning)

Joining

- Nested loop join with index*
- Nested loop join with hashing (also known as hash join)
- Nested loop join with sorted list

Grouping

- Grouping with index* (also known as aggregation)
- Grouping with hashing (also known as hash grouping)

Ordering

- Ordering with index*
- Ordering with sort

* The method directly or indirectly relies on an index for implementation.

Db2 for i also supports parallelism when the optional IBM i licensed feature, Db2 Symmetric Multiprocessing (SMP) is installed. Parallelism is achieved through multiple tasks or threads that work on portions of the query request concurrently. Most, but not all of the implementation methods are parallel-enabled. This allows more systems resources to be used to run the query faster.

Selection

The main job of an index is to reduce the number of physical I/Os that the database must perform to identify and retrieve data. This is the first and most important aspect of query optimization on Db2 for i. The sooner a row can be eliminated, the faster the request will be. In other words, the fewer number of rows the database engine has to process, the better the performance will be (as I/O operations are usually the slowest element in the implementation).

For example, look at the query in Listing 6 that is asking for customer information on orders (where the order was shipped on the first day of the month and the order amount is greater than US $1000):

```
SELECT customer_name, ordernum, orderdate, shipdate, amount
FROM orders
WHERE shipdate IN (‘2008-06-01’, ‘2008-07-01’, ‘2008-08-01’)
      AND amount > 1000
```

Listing 6 - Query with selection predicates

If the table is relatively small, it will not make much difference how the optimizer decides to process the query. The result will return quickly. However, if the table is large, choosing the appropriate access method becomes very important. If the number of rows that satisfy the query is small, it would be best to choose an access method that logically eliminates the rows that *do not match* any ship data values or have amounts greater than $1000. This is where indexes are valuable.

To decide whether or not an index would help, it is important to have an estimate of the number of rows that satisfy this query. For example, if 90% of the rows satisfy this query, then the best way to access the rows is to perform a full table scan. But if only 1% of the rows satisfy the query, then a full table scan might be very inefficient, resource intensive, and ultimately slower. In this case, a keyed-access method that utilizes an index would be the most effective algorithm.

Figure 6 contains a graphical representation of the decision graph that characterizes when index-based methods perform better than a full table scan. When a small percentage of the rows are being accessed, methods utilizing an index offer the best performance. While table scans are often associated with poor performance, that method is the best performing option when a large percentage of the rows in the table are being processed. Remember that with Db2 for i, full table scans are highly efficient because of the independent I/O subsystems, parallel I/O technology, and very large memory system. Thus, a good understanding of the query and data are key pieces of information to know when deciding whether or not to create an index to boost performance.

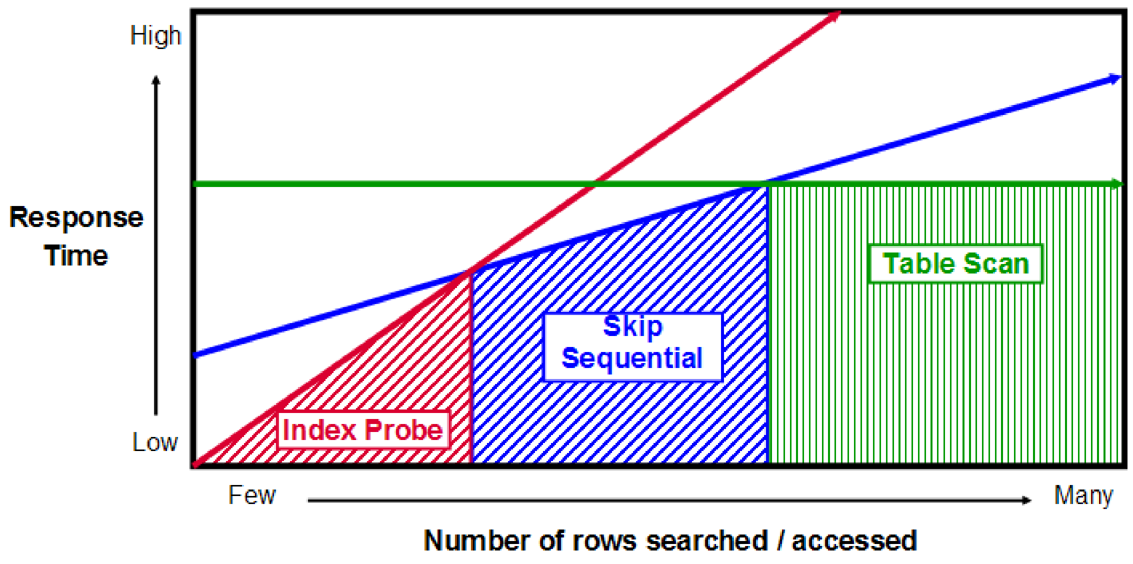


Figure 6 - Performance comparison of access methods

As discussed in the “Statistical usage” section, the optimizer makes this decision based in part, on what it learns from the statistics provided by indexes. Therefore, it is important to have a proper set of indexes defined for each table, regardless of whether the indexes are used for data retrieval or not. In some instances, the optimizer uses the index for optimization, but chooses to implement the query using a non-indexed method.

Nested loop join with index

When Db2 for i builds an access plan for a query that uses *inner join* to access rows from more than one table, it first gathers an estimate for how many rows will be retrieved from each individual table. It then chooses the best access method for each individual table, based on the cost of the various methods available. Based on those estimates and costs, the optimizer then runs through possible variations of the join order, or the sequence in which the tables in the query will be accessed. A query plan is then built that puts the tables in the most cost-effective join order. Unlike some databases, which process a join in the order the tables are provided in the statement, Db2 for i evaluates the possible options and rewrite the query to ensure that the best (least costly) join order is used. For other join types, such as left outer join and exception join, the join must be implemented in the order specified in the SQL request. In other words, the join order cannot be optimized.

As expected, the optimizer needs information from the indexes in order to evaluate the join order. Join order is dependent on the tables that retrieve the most rows as well as the *fan-out* or *fan-in* behavior of the join processing. Join fan-out or fan-in can be simply defined as the number of expected rows that match a given join value as shown. The first join in Figure 7 represents a fan-out scenario. The optimizer assumes that query result set size will be increased by the join because it estimates that each employee is associated with three projects on average. The second join represents a join fan-in condition because not every employee is expected to hold a certification resulting in the query result set size decreasing due to the join. The optimizer estimates the number of rows retrieved by looking at the indexes of the join columns. Therefore, it is very important to build indexes over all of the join columns.

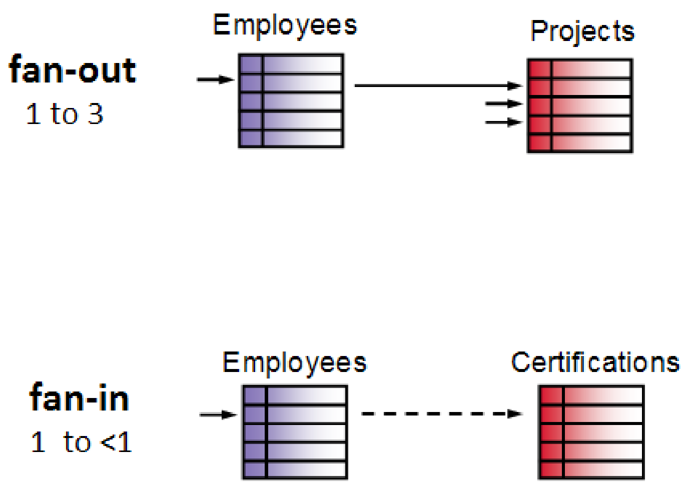


Figure 7 - Join fan-out and fan-in examples

One of the most common methods of join processing is called a nested loop join with index. This method applies to queries where there are at least two tables being joined together, similar to the Employees and Projects table in the example query in Listing 7.

```
SELECT empID, empLastName, projectName
FROM employees A INNER JOIN projects B
    ON A.empID = b.projEmpID
```

Listing 7 - Join query example

As you can see in with nested loop join with index, a row is read from the first table (or dial) in the join using any access method (that is, table scan, index probe + table probe, and so on.). Then, a join key value is built up to probe or look into the index of the second table (or dial). If a key value is found, then the row is read from the table and returned. The next matching key value is read from the index and the corresponding row is read from the table. This process continues until no matching keys are found in the index. The database engine then reads the next row from the first table and starts the join process for the next key value. The nested loop join is not complete until all of the rows matching the local selection are processed from the first table in the join order.

It is important to understand the nested loop join process, so that you can help the implementation to be as efficient as possible. Nested loop join with indexes can produce a lot of I/O if there are many matching values in the secondary dials, or high join fan-out.

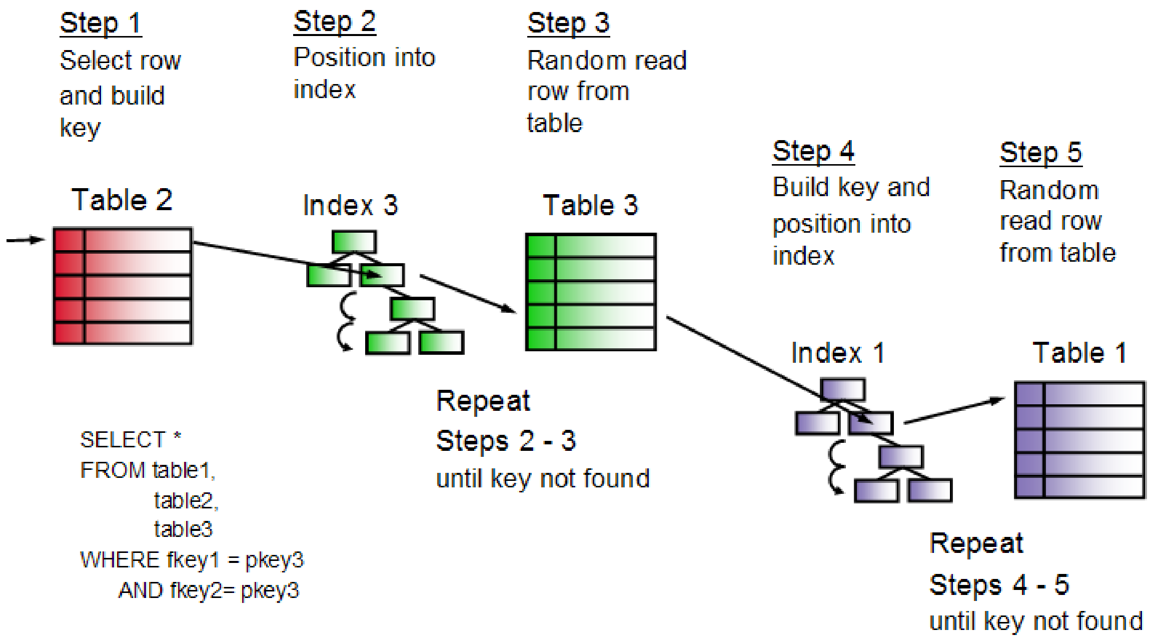


Figure 8 - Nested loop join processing example

Nested loop join with index requires a radix index over the join columns on the tables that are being joined to. If an index does not exist for the join columns in the tables following the first table in the join order, Db2 for i might choose to build temporary indexes over these columns to complete a nested loop join. This temporary index creation or the usage of other temporary data structures to implement the join is one of the most common causes of join performance problems.

With the Db2 for i engine, all nested loop joins with index are processed similar to index probes for local selection. In fact, when both the join and local selection predicates are present for a given join dial, the optimizer can use all of the columns to probe the radix index. This makes the join much more efficient as it narrows down the rows matching the selection and join criteria with a minimum number of I/O requests. This technique can be referred to a multikey join. It is very important to have a radix index available that contains both the local selection column(s) and the join column(s) for a given table. If only the join column is present in the index, the Db2 engine must probe the index for the join key value, then read the table and test the local selection values. If the data does not match the local selection, then the probe of the index and random read of the table is wasted.

Indexes are critical to this process because the database engine can use the index instead of reading the base table. In this way, the optimizer can position to the portion of the index that contains the relevant keys. Assume that the previous join query example is modified to the more complex join, as shown in Listing 8 and the optimizer chooses to process the Projects table first in the join order. If there is an index over the Employees table with key columns deptNum and empID, then the optimizer can use the index to locate only those projects that belong to the specified department (TGZ) and the join key value. This improves performance considerably as it eliminates random reads of the Employees table to process the local selection. Creating an index over the Projects table with key columns **projActive** and **projEmpID** would provide the same performance advantage if the Projects table was placed second in the join order.

```
SELECT empID, empLastName, projectName
FROM employees A INNER JOIN projects B
     ON A.empID = b.projEmpID
     WHERE A.deptNum = 'TGZ' AND B.projActive='Y'
```

Listing 8 - Join query with local Selection example

The SQE query optimizer has the ability to use parallel processing with nested loop joins. Db2 SMP might be used to create a temporary index if required for a nested loop join. Nested loop joins with hashing can also take advantage of parallelism, and do not require an index to perform the join. Joining with a hash table is another join implementation method that uses a hashing algorithm technique to consolidate join values together and to locate the data to be joined. Of course, the user must wait for the hash table to be built and populated before any results are returned from the join.

Grouping and ordering

Other common functions within an SQL query request are grouping and ordering. Using the SQL GROUP BY clause, queries will summarize or aggregate a set of rows together. In Db2 for i, the optimizer can use either an index or a hashing algorithm to perform grouping. The method that the optimizer picks is query, data, and system dependent. In other words, the optimizer will make its selection based on the nature of the query, the amount and type of data, and the system resources available.

When a query includes an ORDER BY clause, the database engine will order the result set based on the columns in the ORDER BY clause. In Db2 for i, the optimizer can use either an index or a sort. Therefore, indexes can be used for this function as well. Sometimes, the ORDER BY clause includes columns already used in the selection and grouping clauses, so the optimizer may take advantage of the “by key” processing used for other parts of the query request. The data is processed in key order, so to speak.

For both grouping and ordering, the optimizer will cost the various methods available, based on the expected number of rows identified in the local selections and join. The optimizer estimates the number of unique groups based on the information that it finds in the indexes. In the absence of indexes and column statistics, the optimizer guesses the number of groups and number of rows per group. As can be imagined, this estimate might be close, but if it is grossly inaccurate, the optimizer might choose an inefficient method for grouping. In working with large business intelligence applications where grouping to construct aggregates is common, there may be millions of groups or millions of rows within a group. As the size of the database scales upward, it becomes even more important for the optimizer to be able to accurately estimate how many rows are involved in a given query operation. Indexes make that possible. In general, hash grouping is most efficient when the query groups a large number of rows per group. In contrast, grouping a small number of rows per group favors index grouping.

Using an index for grouping or ordering can affect the join order of the query. This is true when the grouping and/or ordering columns are from a single table. That table tends to go first in the join order, allowing the database engine to read the rows from the first dial in the join by key, thus allowing the grouping or ordering or both to occur naturally (that is, key order). This may not be the best plan for optimal performance for the entire query. One way to help the optimizer is to create two indexes: one radix index provides selection and join statistics, and the other provides selection and grouping/ordering statistics. Listing 9 contains an example of indexes that would be helpful to the Db2 optimizer for a query that contains both join and grouping criteria. Two different indexes are created over the employees table because you should not assume that the grouping criteria will result in the query optimizer placing the employees table first in the join order. While Db2 for i cannot use both indexes over the employee table for implementation, the optimizer will be able to use the statistics associated with the indexes to gain a deeper understanding of the selectivity of the query, the join fan-out and fan-in, and the grouping/ordering attributes. In turn, the query optimizer will be able to make an intelligent decision on whether or not placing the employees table first in the join order is the best choice from a performance perspective.

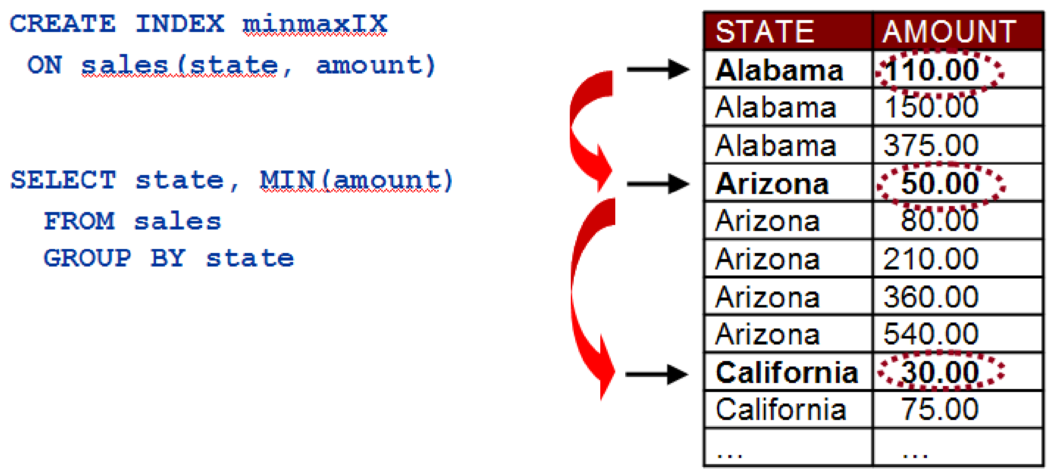
```
CREATE INDEX empJoinIX ON employees(location, empID)
CREATE INDEX empGroupingIX ON employees(location, division, deptNum)
CREATE INDEX projectsJoinIX ON projects(projEmpID)


SELECT A.division, A.deptNum, SUM(B.projectTime)
FROM employees A INNER JOIN projects B
     ON A.empID = b.projEmpID
WHERE A.location='NW'
GROUP BY A.division, A.deptNum
```

Listing 9 - Helpful indexes for query with join and grouping

Another index grouping technique the Db2 for i query optimizer can employ is the Min-Max skipping method. This index-based method enables Db2 to speed up the processing for the MIN and MAX built-in functions. This Min-Max skipping technique is really just another way for Db2 to employ an index probe with multiple keys, and take advantage of the data's ordering with the index. A visual representation of the Min-Max skipping process is shown in Figure 9. The requirement is to have all of the local selection and grouping columns represented in the leading keys of the index followed by the column that is referenced on the MIN or MAX function. Thus, the index in Figure 9 has the **State** column in the leading position of the key definition because the associated SELECT statement contains no local selection criteria, only a grouping reference. With the leading key columns correctly defined along with the **Amount** column, you can see in Figure 9 how Db2 can quickly find the minimum sales amount value for each state because it knows that the first key value for each state contains the minimum sales amount. Instead of having to read though multiple rows to find the minimum sales amount, Db2 is able to find the minimum value with a single index probe operation. Db2 then skips over all of the remaining key values for a state to find the minimum value for the next state. Performance of the MIN function is faster because Db2 is able to bypass the processing of a large number index key values and rows in the underlying table.

Conversely, Db2 for i knows that the last key value for each state will contain the maximum value. Based on this knowledge, the Db2 engine uses a special index probe operation to just position to the last key value for each state.

For grouping requests that include COUNT or SUM functions, the EVI INCLUDE support can speed up performance by having the Count or Sum value be maintained along with the index key.
Listing 1 contains an example of an EVI with a maintained SUM aggregate.



```
CREATE INDEX minmaxIX
 ON sales(state, amount)


SELECT state, MIN(amount)
  FROM sales
  GROUP BY state
```

STATE	AMOUNT
Alabama	**110.00**
Alabama	150.00
Alabama	375.00
Arizona	**50.00**
Arizona	80.00
Arizona	210.00
Arizona	360.00
Arizona	540.00
California	**30.00**
California	75.00
…	…

Figure 9 - Group by Min-Max skipping

Index-only access

The index-only access (IOA) method discussed in the EVI runtime usage section is actually a generic technique that the query optimizer can use with both a radix index and EVI. IOA is a method that the Db2 optimizer can employ when it determines that all of the columns referenced in the query for a particular table are all part of the key definition for an index. The index and query in Listing 10 are an example of such a situation. The query references three columns (deptDivID, deptNum, deptLocation) from the **department** table that also happen to be part of the key definition for index deptIX1.

```
CREATE INDEX deptIX1 ON department(deptLocation, deptNum, deptDivID, deptName)

SELECT deptDivID, deptNum
FROM department
WHERE deptLocation='NW'
```

Listing 10 - Helpful indexes for query with join and grouping

By utilizing the IOA method, the Db2 for i engine will not have to retrieve any data from the department table. Instead, the entire query can be implemented by just processing the index, deptIX1. Normally, index-based access methods require the database engine to perform I/O operations on both the index and table object. Index-only access can offer significant performance improvement by eliminating the I/O requests to the underlying table. The query optimizer is free to use the IOA method for any part of a query – selection, joining, grouping, and ordering.

It is worth noting that the index-only access performance advantage can only be realized on query-based interfaces. Even with the exact same scenario, native record-level access interfaces have no ability to benefit from the performance savings offered by IOA.

Db2 for i indexes and optimization: A summary

Db2 for i makes indexes a powerful tool. Table 2 summarizes the main indexing concepts discussed in this section and compares those concepts for the two types of indexes: radix index and encoded vector index.

	Radix index	**Encoded vector index**
Basic data structure	A wide, flat tree	A symbol table and a vector
Creation interface	OS command(CRTLF), SQL	SQL
Used to enforce constraints (primary key, foreign key, unique)	Yes	No
Used for statistics	Yes	Yes
Used to maintain and provide column aggregates	No	Yes
Used for selection	Yes	Yes, through dynamic bitmap or RRN list
Used for joining	Yes	No, but can be joined to
Used for grouping	Yes	Yes, if data is in symbol table
Used for ordering	Yes	No
Used for index-only access	Yes	Yes

Table 2- Radix and encoded-vector index comparison

Indexing strategies for performance tuning

Now that you understand the basics of Db2 indexing technology, let’s consider how to use this technology most effectively.

There are two approaches to index creation: proactive and reactive. As the name implies, proactive index creation involves anticipating which columns will be most often used for selection, joining, grouping, and ordering; and then building indexes over those columns. In the reactive approach, indexes are created based on optimizer feedback, query implementation plan, and system performance measurements.

In practice, both methods will be used iteratively. As the numbers of users increase, more indexes are useful. Also, as the users become more adept at using the application, they might start using additional columns that will require more indexes. As you create indexes for complex queries, you may find that the initial index created is not used by the query optimizer. Further examination of the query plan, runtime environment, and underlying database objects is probably needed to make adjustments to the index key definition.

Often it works best to use proactive approach for index creation as a starting point, and then add or delete indexes reactively after user and system behavior have been monitored. A later section discusses Db2 for i tooling that can help identify which indexes to add and delete.

Starting point

It is useful to initially build indexes based on the database model and the application(s), in lieu of creating the indexes based on the needs of any particular query. As a starting point, consider designing basic indexes founded on the following criteria:

- Primary, unique, and foreign key columns based on the database model
- Commonly used local selection columns, including columns that are dependent, related or correlated
- Commonly used join columns other than primary or foreign key columns
- Commonly used grouping columns
- Commonly used ordering columns

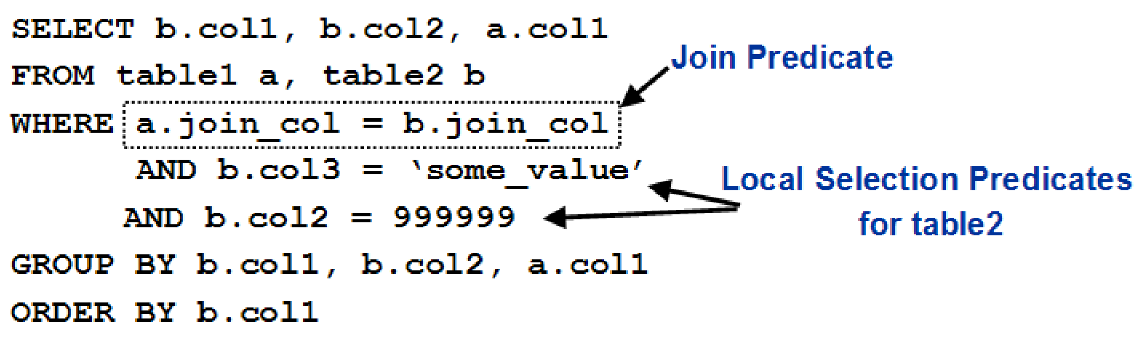
After analyzing the database model, consider the database requests from the application and the actual SQL statements that might be executed. A developer can add to the basic index design and consider building some *perfect* indexes that incorporate a combination of the selection, join, grouping, and ordering criteria. A *perfect* index is defined as a radix index that provides the optimizer with useful and adequate statistics, and multiple implementation methods — taking into account the entire query request.

Proactive approach

Returning to the analogy of the recipe, the goal of an indexing strategy is to give the query optimizer the following two items:

- Information about ingredients or the data contained within the tables, such as the number of distinct values, the distribution of data values, and the average number of duplicate values.
- Choices about which cooking instructions to assemble, or which methods to use to process the query. In many recipes, the cooking method could be steaming, frying, or broiling. The choice depends on the desired result. In the same way, the optimizer has different methods available and will pick the appropriate method based on what it knows about the available ingredients and the desired result.

Before beginning the proactive process, the database model and a set of sample queries are needed. These queries will generally have a format, as shown in Figure 10.



```
SELECT b.col1, b.col2, a.col1
FROM table1 a, table2 b
WHERE a.join_col = b.join_col
       AND b.col3 = 'some_value'
      AND b.col2 = 999999
GROUP BY b.col1, b.col2, a.col1
ORDER BY b.col1
```

Figure 10 - Sample query for indexing strategy discussion

With a similar query in place, the proactive index creation process can begin. The basic rules are as follows:

- Custom-build a radix index for the largest or most commonly used queries. For the example query in Figure 10 that would involve creating two radix indexes over the join columns, a.join_col and b.join_col, and two radix indexes to cover the local selection columns, b.col2 and b.col3. When creating indexes to cover the local selection predicates, it is best practice to only create indexes over those columns that are likely to be referenced in the search predicates for other queries.
- Build single-key EVIs over the local selection columns for ad-hoc, OLAP environments or less frequently invoked queries. Creating an EVI over a local selection column is only recommended for columns that are not unique and contain a relatively low number of distinct values (that is, low cardinality). Using the example query in Figure 10, this recommendation would result in two encoded vector indexes being created over the local selection columns, b.col2 and b.col3.

Clearly, these are general rules whose specific details depend on the environment. For example, the most commonly used queries can consist of three or 300 queries. How many indexes that are built in total by this process will depend on user response time expectations, available disk storage, and the cost of index maintenance?

Perfect radix index guidelines

In a perfect radix index, the order of the columns is important — even making a difference as to whether the optimizer uses the index for data retrieval at all. As a general rule, order the key columns for an index in the following way:

- Equal predicates first. That is, any predicate that uses the **=** operator may narrow down the range of rows the fastest and should therefore be first in the index.
- If all predicates have an equal operator, then order the columns as follows:
  - Selection predicates + join predicates
  - Selection predicates + group by columns
  - Selection predicates + order by columns
  - Order by columns + selection predicates

In addition to the guidelines, in general, the most selective key columns should be placed first in the index. Assume that a query has local selection that references the part_id and part_type columns and that part_id is the primary key column for the parts table. Based on this information, the partid would be listed first in the key definition because it will only select one row while the part_type column most likely selects 5 to 20% of the rows.

A radix index can be used for selection, joins, ordering, grouping, temporary tables, and statistics. When evaluating data access methods for queries, it is best to create radix indexes with keys that match the query's local selection and join predicates. A radix index is the fastest data access method for a query that is highly selective and returns a small number of rows. Columns that are part of a hierarchy should be listed in proper order. For example, it is likely that columns year, quarter, month, day are used together, and separately. An index created with key columns in order (year, quarter, month, day) can support multiple queries that reference part or all of the hierarchy in the WHERE clause.

As stated earlier, when creating a radix index with composite keys, the order of the keys is important. The order of the keys can provide faster access to the rows. The order of the keys should normally be local selection and join predicates, or, local selection and grouping columns (equal operators first and then inequality operators). Radix indexes should be created for predetermined queries or for queries that produce a standard report. A radix index uses disk resources; therefore, the number of radix indexes to create is dependent upon the system resources, size of the table, and query optimization.

The following examples illustrate the guidelines for perfect radix indexes:

Radix example 1: A one-table query

The SELECT statement in Listing 11 uses the **items** tables and finds all the customers who returned orders in the specified year and quarter that were shipped with air freight services. It is assumed that longstanding customers have the lowest customer numbers.

```
SELECT custNum, itemNum FROM items
WHERE year=2007 AND quarter=4 AND
      returnflag='R' AND shipmode='AIR'
ORDER BY custNum, itemNum
```

Listing 11 - Query for radix example-1

The query has four local selection predicates and two ORDER BY columns. Following the guidelines, the perfect index would put the key columns covering the equal predicates first (year, quarter, returnflag, shipmode), followed by the ORDER BY columns, custNum and itemNum.

To determine how to order the key columns covering equal local selection predicates, evaluate the other queries that will be running. Place the most commonly used columns first or the most selective columns first, or both, based on the data distribution.

Radix example 2: A three-table query

Star schema join queries use joins to the dimension tables to narrow down the number of rows in the fact table to produce the result set in the report. The join query in Listing 12 finds the total first quarter revenue and profit for two years for each customer in a given sales territory.

```
SELECT t3.year, t1.customer_name, SUM(t2.revenue_wo_tax), SUM(t2.profit_wo_tax)
FROM cust_dim t1, sales_fact t2, time_dim t3
WHERE t2.custkey = t1.custkey
  AND t2.timekey = t3.timekey
  AND t3.year = 2010
  AND t3.quarter = 1
  AND t1.continent = 'NORTH AMERICA'
  AND t1.country = 'UNITED STATES'
  AND t1.region = 'CENTRAL'
  AND t1.territory ='FIVE'
GROUP BY t3.year, t1.customer_name
ORDER BY t1.customer_name, t3.year
```

Listing 12 - Query for radix example-2

This query has two join predicates and six selection predicates. The first task is to focus on the selection predicates for each table in the query.

The query specifies two local selection predicates for the TIME_DIM table. As a result, the perfect radix index for this table would contain YEAR and QUARTER in the leading part of the key definition, followed by the join column, TIMEKEY.

For the CUST_DIM table, the query specifies four local selection predicates. These predicates are related to each other along a geographical hierarchy (territory-region-country-continent). Because all of the predicates are equal predicates, the order of the index keys for these predicates should follow the hierarchy of the database schema. The index over the customer dimension table should contain TERRITORY, REGION, COUNTRY, CONTINENT followed by the join predicate column CUSTKEY.

The fact table, SALES_FACT, has columns only referenced as joined predicate. Based on this information, the guidelines recommend creating two radix indexes over the respective join columns, TIMEKEY and CUSTKEY. Creating an index over each join columns enables the optimizer to obtain statistics and to examine all of the possible join orders.

According to the guidelines, an index should also be created to address columns referenced on the grouping and ordering clauses (YEAR and CUSTOMER_NAME). However, this query has grouping and ordering clauses that reference columns from more than one table. When this condition occurs, the Db2 for i query optimizer and database engine cannot utilize an index to assist with the grouping or ordering processing. Thus, creating an index to cover the grouping and ordering columns in Listing 12 provides no value.

You can find more information on star schema join optimization in the paper, *Star Schema Join Support within Db2 for i*, at: **ibm.biz**/db2iPapers

Radix example 3: Query with non-equal predicates

Queries containing non-equal predicate operators (>, >=, <, and so on) tend to return more rows than predicates with equality operators. For example, when a query requests all the orders where the date is within a certain range, such as the beginning and end of a quarter, the query may return more rows than a query which retrieves orders for a specific day or week. Because an inequality predicate implies a range of values instead of a specific value, the optimizer makes different decisions about how to build the access plan.

Listing 13 contains a query quite similar to the SELECT statement in Listing 12. The one difference is that the YEAR local selection predicate contains a non-equal predicate operator (<). Accordingly, this changes the index recommendation for the TIME_DIM table that is associated with this non-equal selection predication. Instead of building an index, where the two local selection columns (YEAR and QUARTER) are first in the key order followed by the join column (TIMEKEY), the perfect index for the query in Listing 13 would have a key order of QUARTER, TIMEKEY, and YEAR. Because equal predicates provide the most direct path to the key values, the YEAR column referenced by the non-equal predicate is moved after the columns referenced by equal predicates. This key column ordering produces a logical range of key values that the Db2 for i engine can position to and process contiguously.

```
SELECT t3.year, t1.customer_name, SUM(t2.revenue_wo_tax), SUM(t2.profit_wo_tax)
FROM cust_dim t1, sales_fact t2, time_dim t3
WHERE t2.custkey = t1.custkey
  AND t2.timekey = t3.timekey
  AND t3.year < 2009
  AND t3.quarter = 1
  AND t1.continent = 'NORTH AMERICA'
  AND t1.country = 'UNITED STATES'
  AND t1.region = 'CENTRAL'
  AND t1.territory ='FIVE'
GROUP BY t3.year, t1.customer_name
ORDER BY t1.customer_name, t3.year
```

Listing 13 - Query for radix example-3

Perfect encoded vector index guidelines

Encoded vector indexes are primarily used for local selection on a table. An EVI also provides the query optimizer with accurate statistics regarding the selectivity of a given predicate value. An EVI cannot be used for grouping or ordering and has very limited usage for joins. When running queries that contain joins, grouping, and ordering; a combination of radix and encoded vector indexes may be leveraged in the query implementation. As demonstrated in Figure 6, radix indexes usually offer better performance when the percentage of rows being accessed or selected is relatively small. When the number of rows selected is in the 20% to 70% range, table probe access using a bitmap or RRN list derived from an index is usually the best performance choice.

Remember that the Db2 for i optimizer has the ability to use more than one index to help with selecting the data. This technique may be used when the local selection predicates contain AND or OR operators and a single index does not contain all the proper key columns or a single index cannot meet all of the conditions (for example, OR predicates). Single key encoded-vector indexes can help in this scenario as well because the bitmaps or RRN lists created from the EVIs can be combined to narrow down the selection process.

In general, performance best practices dictate only creating EVIs over column(s) that have a low number of distinct values (that is, low cardinality). Before creating EVIs, refer to the “EVI maintenance considerations” section for a full understanding of the EVI maintenance requirements.

EVI example 1: A one-table query

The SELECT statement in Listing 14 queries the ITEMS table to find all of the customers who returned orders at year end 2010 that were shipped via air. It is assumed that longstanding customers have the lowest customer numbers.

```
SELECT custNum, itemNum FROM items
WHERE year=2010 AND quarter=4 AND
      returnflag='R' AND shipmode='AIR'
ORDER BY custNum, itemNum
```

Listing 14 - Query for EVI Example-1

The query has four local selection predicates and two ordering columns. Following the EVI guidelines, four single key EVIs would be created with key columns covering the equal predicate columns (YEAR, QUARTER, RETURNFLAG, SHIPMODE). The query optimizer will determine which of these EVIs to use for generating dynamic bitmaps or RRN lists. If more than one EVI is chosen by the optimizer, the bitmaps will be logically merged together to identify the rows that the table probe access method needs to retrieve from the ITEMS table.

The EVIs created for this example cannot help sort the data. Thus, the query optimizer will need to add a sort to the query plan in order

EVI example 2: A one-table query with uncovered predicates

Listing 15 contains a query that retrieves all of the courses offered in a specific location (NY) or associated with the specified topic (SQL) under a specific price threshold. When a query that is similar to this one contains two selection predicates that are combined together with the index ORing support, the single key EVI approach provides the query optimizer with a fast access method for identifying the combined set of rows that meet the specified search criteria.

```
SELECT courseNum, courseTitle FROM courses
WHERE (courseLoc = 'NY' OR
       courseTopic = 'SQL')
      AND courseFee < 1000
```

Listing 15 - Query for EVI example-2

The SELECT statement has two local selection columns with equal predicates (courseLoc, courseTopic) that should be covered with single key EVIs. With the equal predicate EVIs in place, the Db2 for i engine can dynamically generate bitmaps or RRN lists that can be logically merged together with index ORing technology to identify the candidate set of course. Because the courseFee column is not covered by an index, the local selection on the courseFee column would be satisfied by reading and testing the fee column in the candidate set of courses identified by the EVIs.

Proactively tuning many queries

The previous examples for both radix and encoded vector indexes assume that an index is being built to satisfy a particular query. In many environments, there are hundreds of different query requests. In business intelligence and data warehousing environments, users have the ability to modify existing queries or even create new ad-hoc queries. For these environments, it is not possible to build the perfect index for every query.

By applying the indexing concepts previously discussed, it is possible to create an adequate number of radix indexes to cover the majority of problem areas, such as common local selection and join predicates. For ad-hoc query environments, it is also possible to create a set of radix and encoded vector indexes that can be combined using the index ANDing and index ORing support to achieve acceptable response times. The best approach will be to create an initial set of indexes based on the database model, the application and the user’s behavior, and then monitor the database activity and implementation methods.

The "Appendix B - Example queries and indexing strategies" section contains additional example queries along with recommended indexes to create. The purpose of the extra examples is to further demonstrate the concept of proactive index creation.

Reactive query tuning

The reactive approach is very similar to the Wright Brothers’ initial airplane flight experiences. Basically, the query is put together, pushed off a cliff, and watched to see if it flies. In other words, build a prototype of the proposed application without any indexes and start running some queries. Or, build an initial set of indexes and start running the application to see what gets used and what does not. Even with a smaller database, the slow running queries will become obvious very quickly. The reactive tuning method is also used when trying to understand and tune an existing application that is not performing up to expectations.

Utilization of the Db2 for i performance monitoring and analysis tools (explained in the “Tools for Index Analysis and Tuning” section) provides access to the following feedback associated with index usage:

- Any indexes the query optimizer recommends creating to improve query performance
- Any temporary indexes created by the Db2 for i database engine to improve query performance
- Implementation methods that were chosen by the query optimizer

Db2 for i includes an Index Advisor as part of its integrated performance tooling. This tool recommends the creation of permanent indexes that the query optimizer believes will improve the performance of the query being executed.

If the database engine is building temporary indexes to process joins or to perform grouping and selection over permanent tables, permanent indexes should be built over the same columns in an effort to eliminate the overhead associated with the temporary index creation. In rare cases, a temporary index is built over a temporary table, so a permanent index is not possible in this situation.

Understanding the implementation methods used in a query plan allows one to focus on other areas that affect database performance such as: system resources, application logic, and user behavior. For example, seeing that a table scan method was selected by the query optimizer when the perfect set of indexes were in place may result in more memory being added to the system to improve the performance of the table scan method.

Table 3 outlines a few problem scenarios and offers suggestions for interpreting the recommendations of the query optimizer and improving performance.

Situation	**Optimizer recommends**	**Recommended action**
There are no indexes built over the tables specified in the query.	Build an index for local selection and joining	Build an index over the selection, and join columns.
You have built an index over some local selection columns or join columns.	Build an index with more key columns for local selection and joining	Build an index over all the local selection columns that are combined with the AND predicates and used with an equal operator, include the join columns.
You have built an index over all the selection columns and performance is a little better but still not acceptable.	Nothing	Consider any join conditions and build perfect indexes. Consider any complex predicates and build derived key indexes if possible. Use Db2 Visual Explain tool to further understand the query plan and runtime behavior.
You have built the perfect index and the optimizer will not use it.	Nothing	Use the Db2 Visual Explain tool to determine which access method the optimizer selected. The optimizer might have determined that a table scan, hash join, or hash grouping will be the best performing methods.
You have built an index that contains all of the relevant columns but the optimizer does not use it.	Build an index that contains the same columns but lists them in a different order.	Build an index with the keys ordered based on the recommendations. Only leading key columns can be probed.

You have built all the recommended indexes, yet query information still indicates that the optimizer's query estimate is not at all close to the actual query run times.	Nothing	Ensure that the optimization goal is properly set for the application's FETCH behavior. Add the following clause to your SQL statement: OPTIMIZE FOR ALL ROWS[1]
You have a query that contains several inequalities and/or the selection predicates are combined with OR conditions.	Build an index over some of the columns, not all	Build single-column indexes over each column, which will encourage the database to use index ANDing or index ORing support The optimizer does not recommend indexes for OR predicates.

Table 3 - Recommend performance actions

The main idea behind all of these recommendations is to give the optimizer as much information as possible about the tables and columns with which you are working. Remember, with Db2 for i, statistical information can be provided to the optimizer by creating the appropriate indexes.

Tuning one query against many queries

As you proceed through this reactive approach that is an iterative process, you will begin to see how indexes that can tune many queries at one time can be built. Start with one query and tune it. Then look at two or three queries. Find the columns that are used in all of the queries and build indexes over those columns. Picking the right columns and getting them in the right order becomes more intuitive and productive.

Index tuning for multiple table queries

When running queries that join tables, the need for the right indexes becomes even more critical. It is very important to analyze query optimizer feedback to understand the query implementation and whether or not the query response time reflects building temporary indexes.

If the system is building temporary indexes over tables, there is a high probability that it is because the indexes are required to process a nested loop join. Nested loop join with index requires indexes over the join columns. The good news is that the optimizer creates an index to complete the query. The bad news is that the user must wait while the index is created. The performance overhead is even worse with CQE because the index is deleted when the query completes. In addition, if the same query is run again, the temporary index must be recreated by CQE. If 20 users are running the same query, each user will have a temporary index created by CQE. SQE minimizes this extra performance burden with its temporary index support. The SQE temporary index processing contains another layer of

[1]The optimizer may generate a different access plan based on the user's information regarding optimizing for all rows or a subset of rows. Refer to the product documentation on how to use this clause.

sophistication that enables the temporary index to be shared across multiple executions of the same query and across multiple users running the same query. Furthermore, the SQE temporary index can be used by any query on the system – not just the query that caused the temporary index to be created.

If a temporary index is being created over a table, at a minimum, you should build a permanent index over the same key columns as the temporary index.

Be aware that the query optimizer sometimes will rewrite the query or portions of a query in order to build a more efficient access plan. Often this rewrite processing involves rewriting parts of the query into a nested loop join. Thus, do not be surprised to see temporary indexes being created for queries that contain no join predicates. The SELECT statement in Listing 16 is an example of the query that Db2 for i often rewrites to utilize the nested loop join method for the subselect processing. The query rewrite involves joining the two tables on department number.

```
SELECT empName
FROM employee e
WHERE status='PT'
  AND e.deptNum IN (SELECT l.deptNum FROM location l WHERE floor=2)
```

Listing 16 - Query without join predicates

Indexing considerations

After realizing what can be done with indexes that are useful for the application, you also need to consider the more pragmatic aspects of indexes such as: creation techniques, maintenance strategies, and capacity planning issues.

Index creation: Best practices

Before creating indexes, you need to figure out where to create the indexes and how to name them. In terms of location, it is strongly recommended to create the indexes in the same schema (or library) as the underlying table. Following this best practice makes database recovery processing simple and efficient.

Regarding index naming conventions, it is usually best to have part of the index name include some reference to the underlying table. When applicable, you need to consider having part of the index name include a reference to the application or reports that the index was created to support as well as the initials of the user creating the index.

The authorities or permission details for an index object are irrelevant from a query optimization perspective. The query optimizer can choose to use any index as part of the query implementation plan whether or not the end user running the query is authorized to use that object. The Db2 engine has the security credentials to use any database object on the system.

Source management

When creating SQL indexes, it is recommended that some sort of documentation process be used to manage and keep track of the index creation source. Managing index source code ensures that when the database is moved to another system, it is easy to create the same indexes for performance. Some ideas for maintaining the source are:

- Place the SQL in an IBM i source file member and use the **Run SQL Statements** (RUNSQLSTM) command to run the SQL. This approach allows users with change management tools based on source members to also use the same tooling to manage the index source.
- Place the SQL in a PC or IFS stream file and use either the RUNSQLSTM command or System i Navigator — **Run SQL Scripts** interface to run the SQL statement.

Performance settings

When creating an index, there are two attributes that can be specified to improve the runtime performance of an index when it is being used for a query. One of those attributes is the UNIT setting, which is shown in Listing 17. The UNIT attribute enables the placement of Db2 for i index objects onto solid-state drive (SSD) devices. The runtime access of Db2 objects stored on SSDs can be substantially faster than those Db2 objects that reside on traditional disks. Db2 for i indexes that contain data that is frequently and randomly accessed and that is updated infrequently are the best candidates for storing on SSDs.

```
CREATE INDEX sampleix ON mytable(mycolumn) UNIT SSD
CRTLF FILE(MYLIB/MYTAB1) UNIT(*SSD)
```

Listing 17 - Index with UNIT media preference example

The other performance attribute to consider is the In-Memory attribute. Specifying the KEEPINMEM(*YES) parameter value on the CRTLF(Create Logical File) or CHGLF (Change Logical file) system commands, causes the SQL Query Engine to automatically request that the Db2 index be asynchronously brought into memory each time that the index is utilized in the query plan for an SQL request. With the index object in memory, the application will have to spend less time waiting for disk I/O operations. The In-Memory attribute is not honored for non-SQL interfaces or SQL statements processed by CQE.

EVI considerations

Using the WITH *n* DISTINCT VALUES clause as shown in Listing 18 can speed up the creation of an encoded vector index as well as minimize index maintenance overhead. This clause tells Db2 the estimated number of distinct values contained in the index key. The number of distinct values is not a hard limit. For example, if the EVI in Listing 18 contains 4000 distinct order date values, the index will create without any errors and be available for the query optimizer to use.

```
CREATE ENCODED VECTOR INDEX myevi2
         ON orders(order_date) WITH 3650 DISTINCT VALUES
```

Listing 18 - EVI with distinct values example

With an EVI, Db2 uses the distinct value count to adjust the default size of the byte code used in the symbol table. This enables Db2 to use the correct byte code size at the beginning of the index build process instead of having to dynamically discover it during the index creation process.

When a distinct values clause is not included, Db2 assumes that the byte code size for the EVI to be 1 byte. An EVI with a 1-byte code size can accommodate 255 distinct values. When the EVI creation process encounters the 256th distinct key value, Db2 has to restart the EVI build process from the beginning with a 2-byte code size. If the EVI encounters more distinct key value than a 2-byte code can handle, the EVI build process is restarted with the maximum 4-byte code size. These extra restarts of the EVI creation process can noticeably slow the performance of the EVI for larger tables.

After the EVI is created, the WITH *n* DISTINCT VALUES clause minimizes the chances of the EVI byte code size having to be adjusted during normal index maintenance. This adjustment causes the EVI to be taken offline. You can find more information on that in the “Encoded-vector index maintenance considerations” section. If unsure of the number of distinct key values, consider using a number greater than 65,535 to create the maximum 4-byte code.

Improving creation performance

Creating indexes can be a very time-consuming process, especially if the underlying tables are large. On IBM i systems with multiple processors, consider using the Db2 SMP licensed feature to create the indexes in parallel. Db2 for i servers enjoy linear scalability when creating large indexes in parallel. In other words, with two processors, the index will be created in half the time. With four processors, the index will be created in one fourth the time, and on a 24-processor server, the index will be created approximately 24 times faster than on a single processor system. Both radix and EVIs are eligible to be created this way. The optional Db2 SMP licensed feature is required to be installed and enabled to create indexes in parallel.

Db2 for i is the only database system that can create indexes in parallel with:

- Online database
- Non-partitioned data sets
- Bottoms-up process (to keep the tree balanced and flat)
- High degree of key compression
- Linear scalability

Another technique when creating multiple indexes on a server with multiple processors is to create indexes simultaneously, one per processor. In other words, submit multiple index creations to run simultaneously. In this way, each index consumes one processor and multiple indexes can be created at the same time. This works particularly well when the indexes are relatively small and the indexes are being created over the same table.

Comparing the creation of SQL indexes with keyed logical files

As mentioned in the beginning of this paper, there are two interfaces available for creating database objects. Both the SQL CREATE INDEX statement and the IBM i command, CRTLF, can be used to create a radix index. Both interfaces create the same index object, but with different attributes. These attributes can have an effect on query optimization and performance. Regardless of how the index was created, the optimizer will recognize and consider the indexes during optimization.

SQL indexes are created with a 64 K logical page size by default while keyed logical files are usually created with a logical page size of 8 K. Regardless of the logical page size, the overall object sizes of a SQL index and a keyed logical file tend to be equivalent. The larger logical page size can result in more efficient index scans and index maintenance because a larger number of key values are brought into memory each time an index page is processed. These are the key benefits in a query environment. Indexes with larger logical page sizes can have a negative impact on the I/O performance within environments that have undersized or suboptimal memory pools.

Starting with the IBM i 6.1 release, the index logical page size can be controlled with either the Page Size clause on the Create Index statement or the PAGESIZE parameter on the CRTLF command. When the page size is not specified, indexes are created with the 64 K logical page size in the following cases:

- Creating SQL indexes on SQL created tables
- Creating SQL indexes on DDS created physical files
- Creating SQL constraints on SQL created tables
- Creating constraints with the ADDPFCST command on SQL created tables
- Creating temporary indexes during query execution

In some cases, the optimizer will choose to create a temporary index with a logical page size of 64 K instead of using a permanent keyed logical file with a logical page size of 8 K. This occurs because the I/O cost of scanning the 8 K index is higher than the cost of using the 64 K index, even considering the time to create the temporary index. To overcome this behavior, the developer should consider creating a SQL index.

SQL indexes that are journaled will cause the journal receivers to fill up more rapidly. To avoid filling up and regenerating new journal receivers more often, consider increasing the journal receiver size to at least 6.5 GB, and set the receiver size attributes RCVSIZOPT(*RMVINTENT *MAXOPT2) with the CRTJRN or CHGJRN system commands. The IBM i 7.1 release attempts to minimize the impact on journal receivers by employing more intelligent journaling algorithms. These new algorithms write only the local 4 K physical disk page containing the key value change to the journal receiver instead of writing the entire logical page associated with the key change.

In general, you should use the index creation interface that best matches your application development strategy and database environment. For example, if a developer is using SQL for object creation and database access, then use SQL to create the indexes.

Duplicate index considerations

When a new SQL index is created with a key definition and attributes that exactly matches the keys and attributes of an existing index, Db2 creates the new index so that it shares the underlying structure (that is, the access path) of the existing index. This eliminates the possibility of having duplicate SQL index objects that are being maintained. The creation of keyed logical files enables sharing of existing access paths when there is a partial match of the key definitions. You can refer to the IBM i Knowledge Center at **ibm.biz**/db2iPapers for additional details on this topic.

Creating an EVI compared with a radix

Encoded vector indexes can be a powerful tool for speeding up data access in decision support and query reporting environments. Similar to any other tool, however, an EVI should only be used when it is a good fit for both the query and database environment. Also, remember that encoded vector indexes are designed to complement radix indexes, and not replace them.

To ensure the efficient usage of EVIs, they should be created using the following guidelines:

- Single-column key or multiple-column keys that have a relatively small set of distinct values
- Columns referenced on local selection predicates or fact table join columns when using star schema join support
- Read-only tables or tables with a minimum of INSERT, UPDATE, and DELETE activity
- Key columns that have a static or relatively static set of distinct values
- Key columns that are often used to aggregate column counts or sums - using the INCLUDE clause

Estimating the number of indexes to create

It would be good to provide a rule of thumb for an appropriate number of indexes to build for different kinds of schemas and databases. But, similar to many things in the application development world, there is not a simple formula for the appropriate number of indexes. It depends on the size of the tables and the relative size of the updates. It depends on the amount of system resources available for the load and update process. It depends on the maintenance window available.

Keep in mind that business intelligence environments are typically read-only, so index maintenance is only an issue during the periodic load and update processes. In addition, business intelligence applications are more likely to allow ad-hoc queries which, especially when radix indexes are the only option, may require more than 10 or 15 indexes to satisfy all the possible query combinations. The ability to combine indexes with the index ANDing and index ORing technology works well for these types of applications.

Tables in OLTP environments have rows being added, changed and deleted — usually at high velocities. Thus, index maintenance must be considered when tuning queries with an extensive indexing strategy.

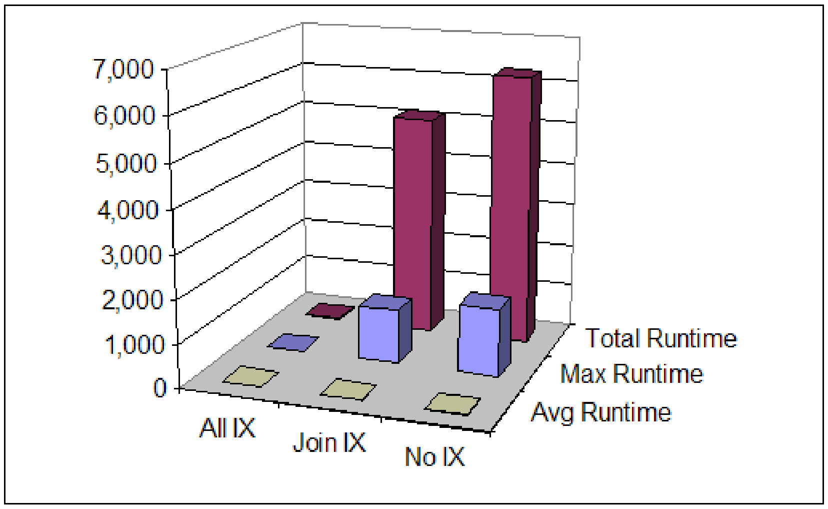
Index maintenance

Index maintenance occurs anytime data is deleted, added, or changed in the underlying table. If a new row is added to a table, any indexes over that table will have to be changed to reflect the new row. Similar processing occurs when rows are deleted or the value of a key column is updated. This index maintenance processing has the potential to decrease the transaction velocity of your application.

The goal of creating indexes for performance is to balance the maximum number of indexes for implementation and statistics while minimizing the number of indexes to maintain. While index maintenance is an important consideration, experience over the years with Db2 for i installations has demonstrated that the benefits of creating a new index for query performance will far outweigh the index maintenance costs. Real-life customer experiences also show that the Db2 for i index maintenance algorithms allow a larger number of indexes to be created and maintained than other RDBMS products. Other RDBMS products have much stricter guidelines on the number of indexes because their index maintenance costs are higher.

To quantify the benefits of adding indexes for performance tuning compared with the costs of index maintenance, IBM constructed and ran performance tests in their labs. Figure 11 summarizes the results of these indexing performance tests. The performance tests involved running a JDBC application against a database consisting of three tables. Two of the tables contained 1.5 million rows and the third table in the test database contained 60 million rows. During the performance tests, the application ran 90 different SQL statements multiple times. These statements consisted of 55 SELECT statements to drive the query workload and 35 data change requests (including insert, update, and delete) to drive index maintenance processing.

The application was run first without any indexes in place, so no overhead from index maintenance. This no index environment produced an average query response time over 20 seconds which is not an acceptable response time for most applications. The next iteration of the tests involved creating four radix indexes to cover the join columns. Despite the additional costs of maintaining four indexes, the new indexes boosted query performance that is enough to reduce the average response time for all the statements by almost 4 seconds. The final iteration of the test involved dropping the join indexes and creating 15 perfect indexes (13 radix and 2 encoded-vector index) to cover most, but not all aspects of the queries (selection, join, grouping, and ordering). Despite increasing the index maintenance costs by almost four times, the average statement running time dropped from 16 seconds to less than one-tenth of a second. It is also easy to understand how these well-tuned query plans consumed fewer system resources (processor, memory, and so on). Accordingly, these more efficient plans free up valuable resources to the benefit of other applications and workloads running on the system.



	Total Time	Max Time	Avg Time
All Indexes	23.547	2.493	0.076
Join Indexes	5,138.851	1,249.081	16.573
No Indexes	6,302.275	1,533.910	20.265

Figure 11 - Indexing performance test results

Different maintenance options

Db2 for i supports three maintenance options for radix indexes: immediate, delay, and rebuild. Encoded vector indexes are always maintained immediately. The immediate maintenance option is the default when creating an index, and it is typically the only option that should be used in a query environment. This is due to the fact that the query optimizer cannot use an index that has a maintenance option of rebuild. In other words, an index with a maintenance option of rebuild is not of any value to the query provider because it cannot be used for statistics or implementation. The optimizer does consider indexes with a maintenance option of delay, but do extra work for these indexes. That is, it looks at how many changes are waiting to be performed on the index by looking at the delayed maintenance log and predicts how these changes will affect the index. Besides making the optimizer guess at how the pending changes will affect the index, using an index with a maintenance option of delay causes additional query execution time — simply because the pending changes must be performed on the index when it is opened. This index maintenance increases the query response time.

Parallel maintenance for insert operations

Another recommendation is to take advantage of parallel-index maintenance using the optional Db2 SMP feature of the IBM i operating system. Parallel-index maintenance enables greater I/O velocities by using multiple database tasks to maintain indexes in parallel during *blocked* insert operations. For example, if there are eight indexes over a given table and applications are inserting data into the table, the database tasks can maintain each index in parallel. Otherwise, the application would wait for each of the eight indexes to be maintained serially. Parallel-index maintenance reduces the response time of the blocked write operation by increasing usage of processor resources during the operation. As a result, the usage of parallel-index maintenance only makes sense on systems that have CPU resources available to consume.

When a Db2 table (or physical file) is reusing deleted rows, Db2 cannot perform blocking at the database engine level even when the application is performing blocked insert operations. This is due to the fact that reuse of deleted rows requires Db2 to extract the individual rows out of the blocked insert, so that each individual insertion can be placed in a deleted row slot. Efficiency and performance is much better when Db2 can insert the new rows as a single block instead of one row at a time. This lack of blocking also prevents the usage of parallel-index maintenance because it can only be employed on blocked insert operations.

While blocking can be enabled by turning off the **reuse deleted rows** option, it is not an ideal solution because the reuse capability provides positive benefits such as eliminating the requirement to reorganize table in order to reclaim the space occupied by deleted rows. As a result, there’s an option on the Override Database File (OVRDBF) command which provides the ability to temporarily enable Db2 row-level blocking and parallel-index maintenance for tables defined with the Reuse Deleted attribute set to *YES. By specifying a parameter value of REUSEDLT(*NO) on the Override command, the job requesting the override will have the table treated by Db2 as if it was created with REUSEDLT(*NO). Batch jobs can be modified to invoke this override in order to speed up their performance as REUSEDLT(*NO) enables the batch process to benefit from both Db2 row-level blocking and parallel-index maintenance. When that same table is accessed by other jobs on the system, the Db2 engine will use the table's permanent **reuse deleted rows** option setting of *YES.

Maintenance options for batch and bulk operations

Depending on the number of rows being changed, the system configuration, and the number of indexes that are over the database table; the recommendation might be to drop all indexes, perform the updates, and then rebuild the indexes upon completion of the process. This is due to the fact that maintenance of indexes in a serial fashion can slow down the batch processes or workloads that load data in bulk.

In general, if the percentage of rows being added, deleted, or changed in a process is more than 30% of the total size of a table, it is probably better to drop the indexes and rebuild them after the update completes. The threshold for when to use parallel-index maintenance instead of dropping indexes varies widely based on the size, number, and complexity of indexes in your database. Also, it is good to remember that parallel-index maintenance is limited to blocked insert operations, and parallel maintenance is not done for delete and update operations. Even if the percentage of rows is not relatively large, both the methods should be tested and benchmarked to determine which method performs better for the workload in question.

EVI maintenance considerations

Encoded vector indexes provide an advanced technology for tuning certain classes of queries. Although the creation of encoded vector indexes may drastically simplify the indexing strategy, it is important to understand the nuances of how Db2 for i maintains an EVI to ensure that maintenance costs can be minimized while maximizing the benefits. Two of the factors that can have a significant impact on EVI maintenance performance are the maximum number of distinct key values and the insertion order of new keys.

Maximum number of distinct values

As discussed in the "EVI considerations" section, the WITH *n* DISTINCT VALUES clause can be specified to improve the performance of EVI creation and maintenance. This clause helps the database engine to determine the correct byte code size up front, so that it does not have to dynamically determine the size through trial and error. Because the EVI symbol table compresses the key values into a 1-byte, 2-byte, or 4-byte code; the maximum number of distinct values for each byte code size is as shown in Table 4.

If WITH *n* DISTINCT VALUES is between	Vector element byte code size
1 and 255	1 byte
256 and 65,535	2 bytes
65,536 and 4.2 billion	4 bytes

Table 4 - EVI number of distinct value ranges

Correct usage of the WITH *n* DISTINCT VALUES clause also minimizes the chances of the EVI being rebuilt during index maintenance because the number of distinct key values exceeds the value threshold for a particular byte code size. For example, if an EVI is created with a 1-byte code and then the 256th distinct key value is added at a later time, Db2 for i automatically rebuilds the index with a 2-byte vector table. While the byte code size adjustment is taken care of automatically, performance degradation occurs while the index is rebuilt because the EVI will not be available to the query optimizer or Db2 engine. Therefore, it is important to have a good idea of how many distinct key values will be represented before creating the EVI.

The consequence of defining more distinct values than are needed is simply that the index will take up a little extra disk space. There is no discernable difference in performance between a 1-byte code and a 4-byte code. For example, if an index with 300 distinct values is defined and only 200 distinct values are ever inserted, the vector will have one extra byte for every row in the database table. Except for organizations with exceptionally constrained disk storage, the extra byte should be insignificant. Given a 1 billion row table, an extra byte would consume roughly 1 GB of space.

Insertion order

When new values are inserted into a table with indexes present, Db2 for i automatically maintains the currency of the indexes. When a new row is inserted into the table, Db2 for i scans the EVI symbol, finds the matching value, and updates the statistics in the symbol table. If a new distinct value is introduced to an existing EVI, one of following two actions occurs:

- If the new value is logically ordered after the last distinct value in the symbol table, the value is added to the end of the symbol table. Assume an EVI has been built over the Year column in a table that contains year values from 2008 to 2010. When rows are added to this table for 2011, an entry for 2011 is added to the end of the EVI symbol table. This insertion scenario does not cause any performance issues.
- If the new distinct key value is not logically ordered after the last distinct value in the symbol table, the key value is placed in an overflow area of the symbol table and the value remains there until the index is rebuilt or refreshed. Consider an EVI created over a Country column for a company that only has customers in the United States, Canada, and Mexico. When the company expands into China, the China key value and it associated statistics are placed in the overflow area of the symbol. The new key value must be placed in the overflow area because United States is the last distinct key value in the symbol table and China is logically ordered before the United States value.

During the refresh process, the EVI is placed in delayed maintenance mode and is not available for use by the optimizer or database engine. Significant performance degradation can be experienced during this process. For this reason, EVIs are usually not recommended for OLTP environments where this insertion pattern occurs. In other words, if the key column has a continuous stream of new distinct values inserted, it is very likely that the keys will arrive in random order, and thus be placed in the overflow area. Also, it is recommended that whenever a large number of distinct values are being inserted by a process, you should consider dropping all the EVIs and recreating them after the insert process has completed. For example, when loading data that contains many new distinct keys into data warehouse tables, it is a good idea to drop the EVIs prior to the load process and recreate the EVIs after the loading process.

Also, remember that the number of key values in the overflow area can be checked by issuing the Display Field Description (DSPFD) system command on the EVI or by right-clicking on any index object with ACS and clicking the **Description** task.

While it is rarely required, there are two ways to refresh the EVI to incorporate the key values stored in the overflow values back into the symbol table:

- Drop the EVI and recreate it.
- Use the CHGLF command with the Rebuild Access Plan parameter set to *Yes (FRCRBDAP(*YES)). If there are keys in the overflow area, this command will only refresh the EVI, not rebuild it. In other words, the underlying table is not required or read during a refresh. A refresh is not as time consuming as dropping and recreating the index, and it does not require any knowledge about how the index was built. This command is especially effective for applications where the original index definitions are not available. If there are no keys in the overflow area, the EVI is rebuilt – this is equivalent to dropping and recreating the index.

You should now understand that EVI usage and maintenance must both be carefully considered, and a proper understanding of data and application behavior is essential. Remember, EVIs are best when the key cardinality is relatively small (with lots of duplicate keys). Table 5 shows a progression of how EVIs are maintained and the conditions under which an EVI is most and least effective, based on the EVI maintenance idiosyncrasies.

	Condition	**Characteristics**
Most effective	When inserting an existing distinct key value	• Minimum overhead • Symbol table key value looked up and statistics updated • Vector element added for new row, with existing byte code
↓	When inserting a new distinct key value — **in order**, within byte code range	• Minimum overhead • Symbol table key value added, byte code assigned, and statistics assigned • Vector element added for new row, with new byte code
Least effective	When inserting a new distinct key value — **out of order**, within byte code range	• Minimum overhead if contained within overflow area threshold • Symbol table key value added to overflow area, byte code assigned, and statistics assigned • Vector element added for new row, with new byte code • Considerable overhead if overflow area threshold reached • Access path invalidated — not available • EVI refreshed, overflow area keys incorporated, new byte codes assigned (symbol table and vector elements updated)
	When inserting a *new* distinct key value — out of bytecode range	• Considerable overhead • Access path invalidated — not available • EVI refreshed, next byte code size used, new byte codes assigned (symbol table and vector elements updated).

Table 5 - EVI maintenance summary

Tools for index analysis and tuning

With a cost-based query optimizer and a broad set of indexing choices, having a good set of performance monitoring and analysis tools available is advantageous. As you can see in Figure 12, Db2 for i provides a wide variety of tools for evaluating what indexes are being used or not used by the optimizer and how to influence its choices. Depending on how much is known at the beginning of the analysis process, different tools may be selected, or a combination of tools and methodologies may be used.

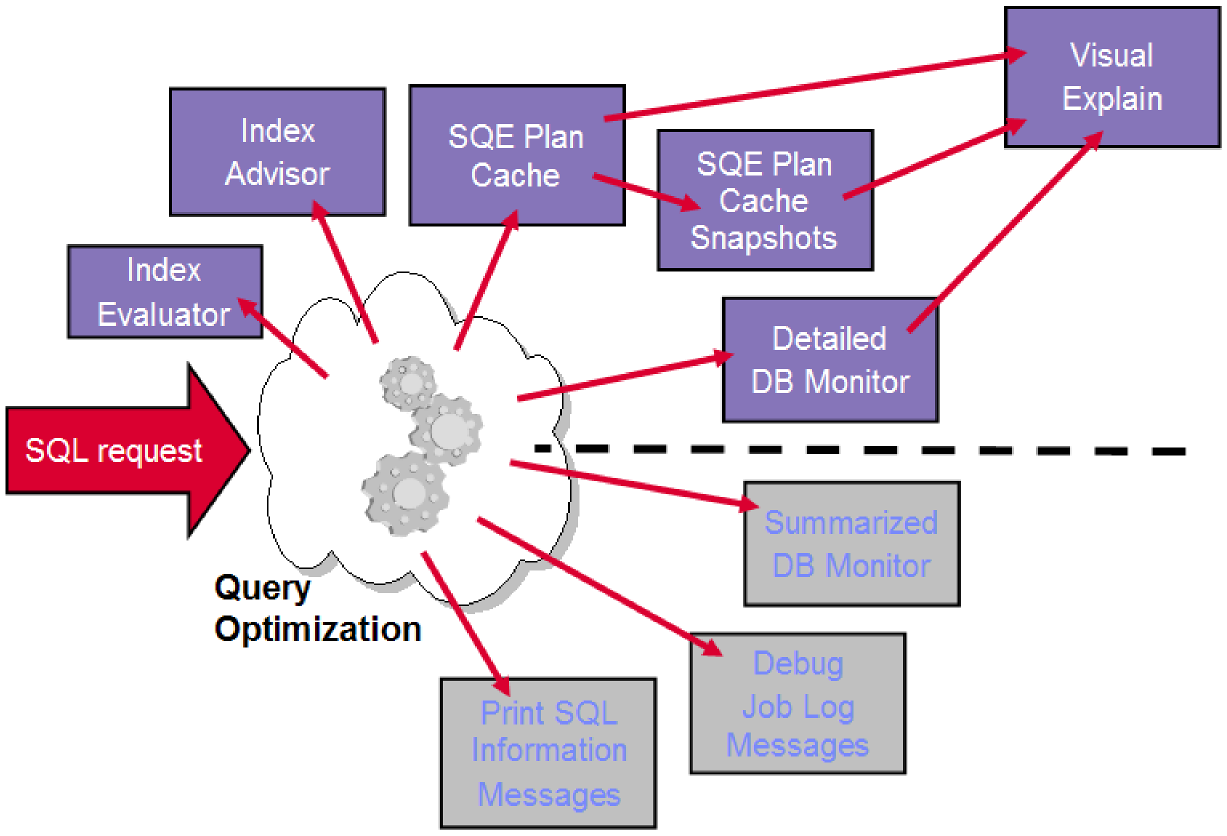


Figure 12 - Query optimization feedback tools

In Figure 12, notice that there are several tools that appear below the dashed line. All of the tools below the dashed line are considered nonstrategic or useless in terms of understanding query plans. Many have not been enhanced by IBM since the V5R2 release when the SQL Query Engine was first introduced. As a result, any users still relying on these tools for feedback from the query optimizer are receiving incomplete or inaccurate information – especially for those queries that are run by SQE. For instance, none of the enhanced index advice provided by the SQE query optimizer is available to users of the nonstrategic tools. Thus, it is critical that users start leveraging the strategic Db2 for i performance tools to be more effective and efficient when analyzing and tuning queries.

The majority of these Db2 for i performance tools are provided with the IBM i Access Client Solutions (ACS). In addition to the performance toolset, this graphical management client also provides interfaces for creating and managing Db2 for i objects as well as running SQL scripts.

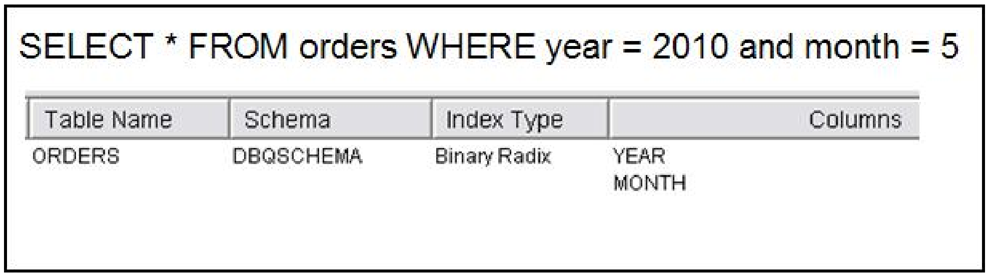
Index Advisor

The Index Advisor is a component of Db2 for i that provides feedback on suggested indexes that the query optimizer believes can help the specified query run faster. The index advice can be accessed and acted on with a number of different tools. Before learning about these tools, it is important to have a thorough comprehension of the index advice itself.

The first thing to understand is that the Index Advisor is recommending the creation of SQL indexes and not keyed logical files. The Index Advisor processing is counting on the performance benefits of the larger logical page size that SQL indexes are created with by default.

An even more important aspect to understand about the Index Advisor feedback is that it is advice. Similar to the advice that one receives on stock market tips, there is no guarantee that creating the suggested index will improve performance. In fact, it is entirely possible that the query optimizer will not even use the index that it recommended creating.

There are several factors that contribute to this conundrum associated with index advice. The primary factor is that the query optimizer has to make assumptions when it advises the creation of indexes. The first assumption that the query optimizer makes is that the column being compared on the selection or join predicate is highly selective. Highly selective means that the specified search criteria returns a low percentage or small number of rows. The sample query and generated index advice in Figure 13 provide a great scenario for understanding this assumption issue. By recommending the creation of an index with key columns of Year and Month in this example, the query optimizer assumes that only a small percentage of the orders occurred in May 2010 (1% of the rows to be exact). Without an index over the search columns, the Db2 optimizer has to guess on the distinct number of year and month values as well as the distribution of values stored within those columns. If the orders table does contain data from several years that validates the optimizer's assumption of high selectivity, then the creation of the advised index will probably result in the optimizer using that new index to improve performance. However, if the orders table contains only data from the second quarter of 2010 and the majority of the orders were placed in May, the optimizer would most likely not use the advised index and performance would remain unchanged. Even in this situation, the advised index provides some value because it provides the query optimizer a deeper understanding of the data values in the columns.



```
SELECT * FROM orders WHERE year = 2010 and month = 5
```

Table Name	Schema	Index Type	Columns
ORDERS	DBQSCHEMA	Binary Radix	YEAR MONTH

Figure 13 - Simple Index Advisor example

This situation is a great reminder why the index advice provided by Db2 for i should not be blindly followed because there will be situations where the Index Advisor's high selectivity assumption is incorrect, resulting in the advised index not being used for data access. The query optimizer will not be able to understand your data and queries as thoroughly as you understand them. Utilize the index advice as a guide to performance tuning and understanding your data, but always validate the index advice before acting on it. Your superior knowledge of the data and SQL statements involved will help ensure that the right set of indexes get created.

Knowledge of how the data is queried and accessed can also help you build indexes that benefit the widest range of queries. The Index Advisor output is based on the analysis of a single query. As a result, the advised key order is the order that is ideal for the query being executed. However, that key order may not be ideal for a larger set of similar queries, or the most frequently run queries. Ideally, the key column order suggested by the Index Advisor should be considered against the most common data access patterns of your applications and end users. The Index Advice Condenser can be helpful in this regard.

The final assumption to consider is that the Index Advisor is assuming that the advised index will be the best performing access method when used with the index probe method. With a cost-based query optimizer, this cannot be validated until the recommended index has actually been created. This assumption is another common cause of situations where creating the advised index provides no performance benefit. The other thing that can happen in this situation is that the creation of the new index results in different indexes being advised by the Index Advisor. This situation occurs because the newly-created index now helps the optimizer better understand the data. Usually this results in a completely different plan being chosen with potentially different indexing requirements.

Index advice details

Both the CQE and SQE query optimizers provide index advice. However, the index advice provided by the SQE query optimizer is much more robust and complete. The differences documented in Table 6 should not be a surprise given that SQE is the strategic database engine which results in the majority of enhancements only being added to SQE.

	SQE Index Advisor	**CQE Index Advisor**
Types of index advised	Radix and EVI	Radix
Analyzed portions of the query	• Local selection predicates • Join predicates • Ordering criteria • Grouping criteria	Local selection predicates (only when table scan and index scan methods are employed)
Multiple indexes advised per table	Yes	No
Supported Db2 performance tools	• Detailed SQL Monitor • SQE Plan Cache • SQE Plan Cache snapshot • Visual Explain • System-wide Index Advisor	All tools

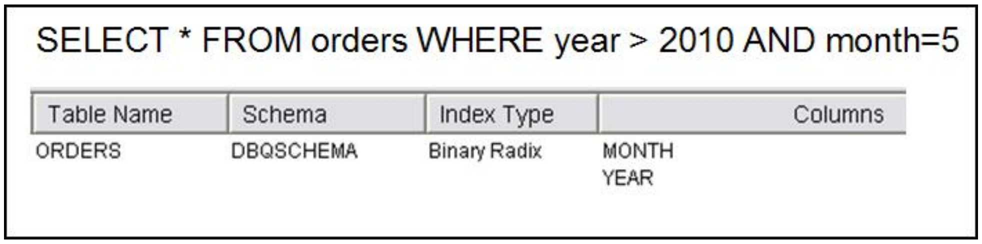
Table 6 - SQE and CQE index advice comparison

The first difference to notice is that only the SQE Index Advisor is capable of recommending the creation of an EVI. This is a minor difference when compared with the portions of the query analyzed by the two index advisors. The CQE Index Advisor comes no where close to advising perfect indexes because the index advice is based solely on the selection predicates within a query. In addition, the CQE Index Advisor only provides index advice when the query optimizer has selected a table scan or index scan method. If the query optimizer selects to use a less-than-ideal index for data access, then a better index will not be suggested by the CQE Index Advisor. The Visual Explain tool does try to make up for some limitations of the CQE Index Advisor. Refer to the “Temporary index feedback” section for additional details.

When an advised index contains multiple key columns, the key columns are generally ordered in the following manner to enable index probe operations.

- Without join: location selection column(s)
- With join: location selection columns(s), join column(s)
- With grouping columns from one table: location selection column(s), grouping column(s)
- With ordering columns from one table: location selection column(s), ordering column(s)

For queries containing local selection or join predicates, the columns involved in equal comparisons are listed first, followed by a single column from nonequal comparisons. Having a single column from a nonequal comparison still enables the advised index to be used for index-probe operations. The index advice ordering of the columns in equal predicates being ordered before the columns associated with nonequal comparisons is demonstrated in Figure 14. The simple query from Figure 13 has been modified to perform a *greater than* comparison against the year column. Notice that this simple change alters the advised index to list the month column first in the key order because it is the only predicate with an equal comparison. The year column occupies the second position.



```
SELECT * FROM orders WHERE year > 2010 AND month=5
```

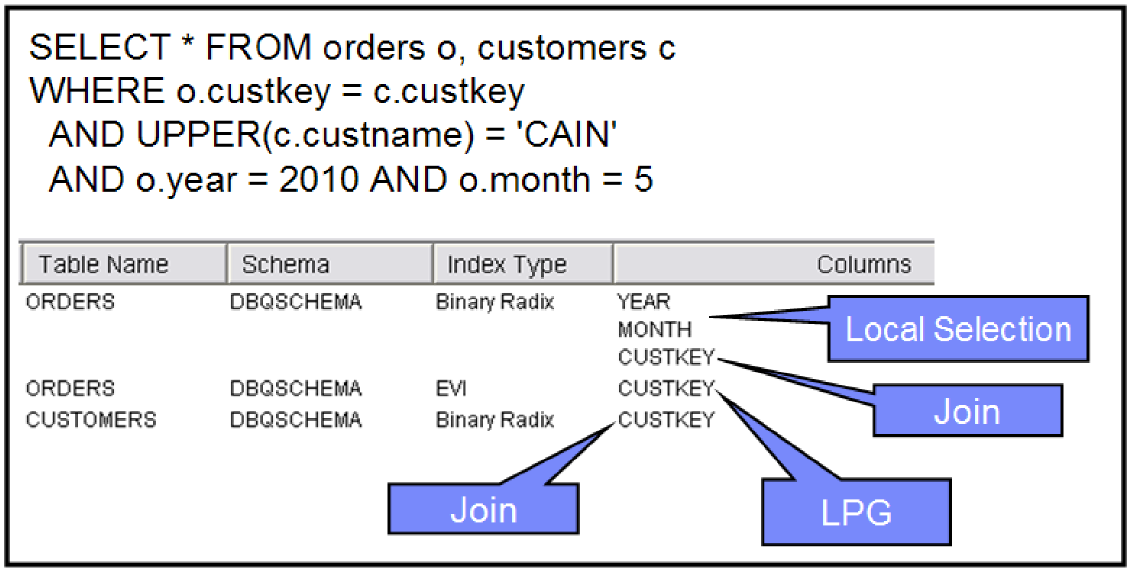
Table Name	Schema	Index Type	Columns
ORDERS	DBQSCHEMA	Binary Radix	MONTH YEAR

Figure 14 - Index Advisor example with nonequal comparison

Figure 15 contains an example of a more complex query along with the generated index advice. The generated index advice demonstrates the unique ability of the SQE Index Advisor to generate complete index advice. The advised radix index for the Orders table contains key columns of Year, Month, and Custkey that includes both the local selection predicates as well as the join column for the Orders table. In contrast, the CQE Index Advisor would only recommend an index with keys of Year and Month for this query.

The example in Figure 15 also validates SQE Index Advisor's capacity to advise multiple indexes for a single table. Notice that an EVI with a single key of Custkey is also advised for the Orders table. This EVI advice is driven by a capability of the SQL Query Engine known as look-ahead predicate generation (LPG). LPG enables the SQE query optimizer to transfer local selection predicates from one table to another. In this example, the SQE Index Advisor is suggesting that the Custname selection predicate on the Customers table can be transferred to the Orders table. The selection transfer occurs through the join predicate (o.custkey = c.custkey). The LPG technology can make some of the index advice difficult to understand at first glance, so it is good to be aware that LPG influences the SQE Index Advisor. You can find a detailed discussion of LPG in the white paper, *Star Schema Join Support within Db2 for i* at: **ibm.biz**/dbiPapers

The outputted index advice in Figure 15 contains no index advice for the column custname. Although this column is referenced on an equal selection predicate, the UPPER function invocation prevents index advice. While SQE can use a derived index with a key definition of UPPER(custname), the SQE Index Advisor does not have the ability to advise indexes for predicates involving function calls. This is one of the Index Advisor deficiencies explained in the “Index advice limitations” section.



```
SELECT * FROM orders o, customers c
WHERE o.custkey = c.custkey
  AND UPPER(c.custname) = 'CAIN'
  AND o.year = 2010 AND o.month = 5
```

Table Name	Schema	Index Type	Columns
ORDERS	DBQSCHEMA	Binary Radix	YEAR MONTH CUSTKEY
ORDERS	DBQSCHEMA	EVI	CUSTKEY
CUSTOMERS	DBQSCHEMA	Binary Radix	CUSTKEY

Figure 15 - Complex Index Advisor example

Index advice limitations

There are some query predicates and indexing technology that are not supported by either the SQE or CQE Index Advisor. This is a list of the current limitations:

- The advisors cannot provide index recommendations for OR predicates that reference different columns. For example, no index advice is returned for the following search predicate, SIZE='L' OR COLOR='Blue', because the predicates combined with the OR operator reference different columns. Creating a single key index over each column (Size and Color) can improve the performance for this combination of search predicates because the query optimizer can use index ORing technology. When the OR predicates reference the same column (SIZE='L' or SIZE='M'), index advice will be provided. Similarly, index advice is provided for IN predicates, SIZE IN ('L','M').
- Indexes with derived key columns are not recommended by the Index Advisor. A search condition such as UPPER(company_name)='ACME' will never cause index advice to be generated even though an index with a derived key definition of UPPER(company_name) would provide the query optimizer with a usable index.
- The EVI INCLUDE support enables aggregation values to be stored and maintained within the EVI symbol table. With these aggregations in place, Db2 can quickly return the aggregated values for a query such as the following, SELECT region, SUM(salesamt) FROM sales

GROUP BY region. The Index Advisor support does not provide recommendations for aggregation functions that would benefit from the EVI INCLUDE support.

System-wide Index Advisor

Prior to V5R4, the index advice could be accessed only when a tool such as the database monitor had been manually activated prior to the query or application being run. Even if this was done, an analyst had to deal with the complexities of extracting the advised index details of various job logs and database monitor files. To eliminate these complexities, the Index Advisor was enhanced so that it is always running, making the index advice available on demand on any system. This capability is known as the System-wide Index Advisor. With this support, accessing index advice is as simple as a mouse click. You just need to right-click the database name within the ACS Schemas tool and click the **Index Advisor** task, as shown in Figure 16, and you will be given the output as displayed in Figure 17. The System-wide Index Advisor data can be filtered to report on a schema or table-level by right-clicking on those objects and clicking the **Index Advisor** task.

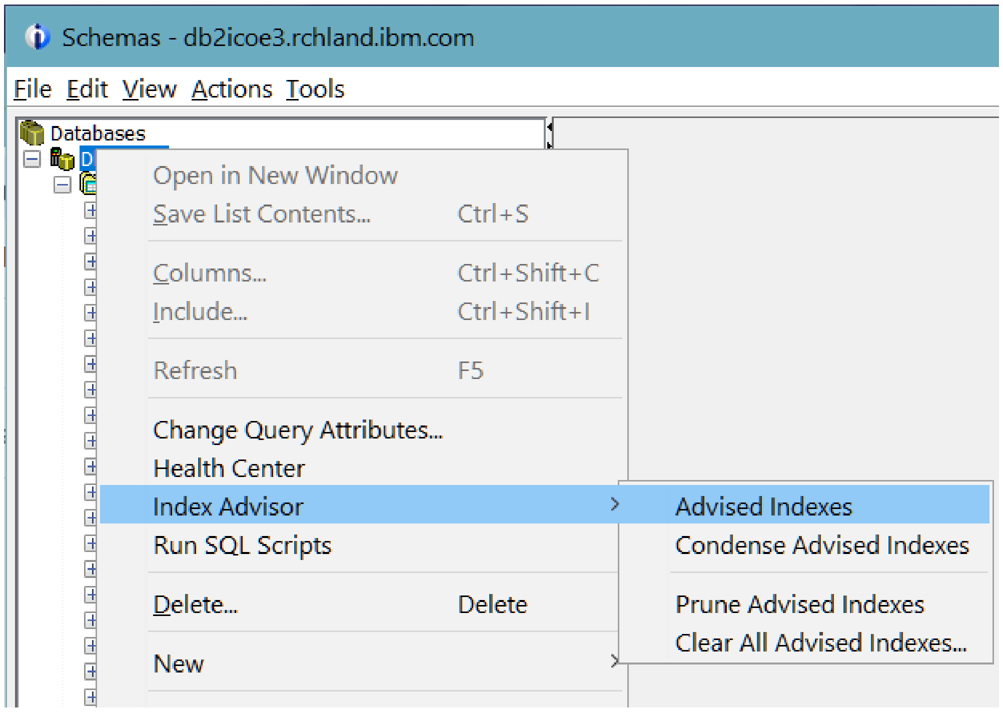
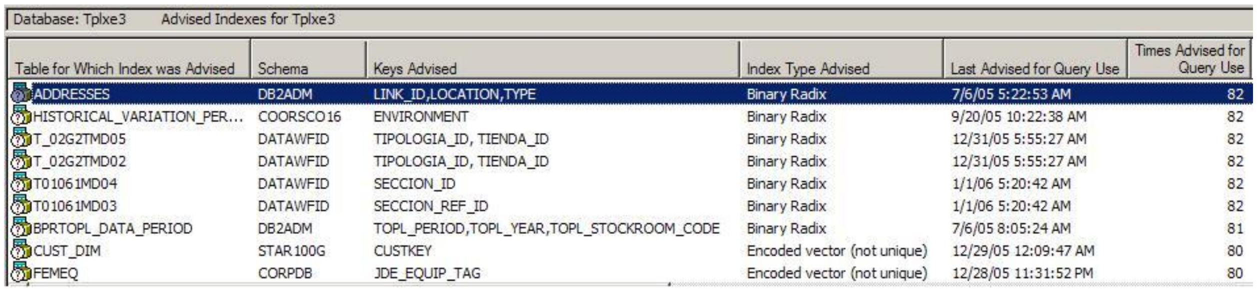


Figure 16 - System-wide Index Advisor interface

The index advice data shown in Figure 17 is a customized view of the output window. The graphical interface allows the columns to be reordered or removed from the view. With the default layout, you need to manually scroll to the right-hand side of the output window in order to view the contents of the columns containing the number of times an index was advised along with the last time the index was advised. This interface also makes it easy to create the recommended index. You need to simply right-click an index to create it. However, remember that it is not considered as a best practice to blindly create the advised indexes from this interface. Ideally, your database and queries should also be reviewed before acting on the index advice.



Database: Tplxe3 Advised Indexes for Tplxe3

Table for Which Index was Advised	Schema	Keys Advised	Index Type Advised	Last Advised for Query Use	Times Advised for Query Use
ADDRESSES	DB2ADM	LINK_ID,LOCATION,TYPE	Binary Radix	7/6/05 5:22:53 AM	82
HISTORICAL_VARIATION_PER...	COORSCO16	ENVIRONMENT	Binary Radix	9/20/05 10:22:38 AM	82
T_02G2TMD05	DATAWFID	TIPOLOGIA_ID, TIENDA_ID	Binary Radix	12/31/05 5:55:27 AM	82
T_02G2TMD02	DATAWFID	TIPOLOGIA_ID, TIENDA_ID	Binary Radix	12/31/05 5:55:27 AM	82
T01061MD04	DATAWFID	SECCION_ID	Binary Radix	1/1/06 5:20:42 AM	82
T01061MD03	DATAWFID	SECCION_REF_ID	Binary Radix	1/1/06 5:20:42 AM	82
BPRTOPL_DATA_PERIOD	DB2ADM	TOPL_PERIOD,TOPL_YEAR,TOPL_STOCKROOM_CODE	Binary Radix	7/6/05 8:05:24 AM	81
CUST_DIM	STAR100G	CUSTKEY	Encoded vector (not unique)	12/29/05 12:09:47 AM	80
FEMEQ	CORPDB	JDE_EQUIP_TAG	Encoded vector (not unique)	12/28/05 11:31:52 PM	80

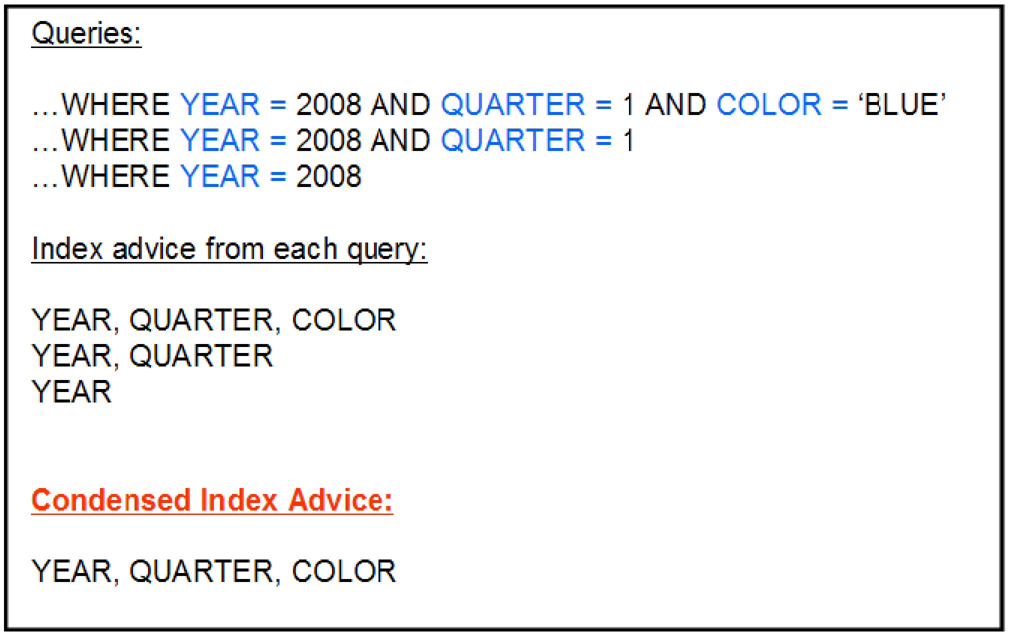
Figure 17- System-wide index advisor output

The output window produced by the system-wide index advisor is a graphical representation of the index advice data that the SQE and CQE Index Advisors log into the SYSIXADV table in the QSYS2 schema. This simple table repository design means that the index advice can also be accessed programmatically by simply querying the SYSIXADV table.

Index Advice Condenser

As index recommendations are logged into the system-wide index advisor repository, there is no analysis performed to determine whether or not a previously advised index with similar keys would meet the requirements of the index being advised. The Index Advisor only looks for matching index advice in order to determine if a new entry needs to be added to the repository or the metrics for an existing index recommendation need to be updated. Thus, there is a good probability that the system-wide index advisor will return index advice that is somewhat redundant in terms of indexes with overlapping key definitions.

As a result, an Index Advise Condenser is available to analyze all of the index advice as a whole and return just the condensed index advice by eliminating any index that is a subset of another advised index. The condensed index advice capability is best understood by looking at an example. In Figure 18, you can see that there are three queries run against the same table resulting in three unique sets of index advice being generated for combinations of the Year, Quarter, and Color columns. By default, these somewhat overlapping sets of index advice are all returned by the System-wide Index Advisor.



Queries:

…WHERE YEAR = 2008 AND QUARTER = 1 AND COLOR = 'BLUE'
…WHERE YEAR = 2008 AND QUARTER = 1
…WHERE YEAR = 2008

Index advice from each query:

YEAR, QUARTER, COLOR
YEAR, QUARTER
YEAR

Condensed Index Advice:

YEAR, QUARTER, COLOR

Figure 18 - Condense index advice example

In contrast, the Index Advice Condenser would analyze this set of index advice and reduce the index advice to the single index recommendation, which is listed at the bottom of Figure 18. The key column ordering in the condensed index advice allows the recommended index to provide index probe access for all the three queries that originally generated the index advice.

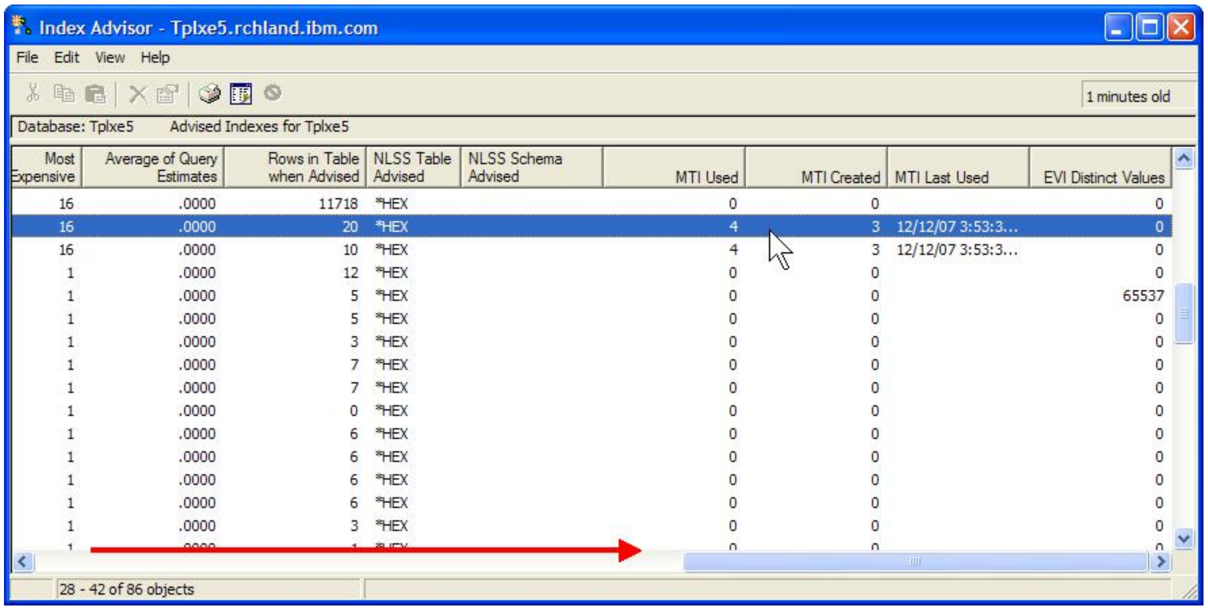
The condensed index advice is helpful for speeding up index analysis because it reduces the number of advised indexes that need to be reviewed. The Condenser is accessed by selecting the **Condense Advised Indexes** task, shown in Figure 16.

The QSYS2 schema contains a CONDENSEDINDEXADVICE view. Usage of this view enables the condensed index advice to be access programmatically.

Autonomic indexes

Starting with V5R4, the SQE query optimizer can improve performance by automatically acting on the recommendations provided by the SQE Index Advisor. In some situations, the SQE optimizer will decide to create a temporary index containing the keys from the index advice. These temporary indexes are known as autonomic indexes because SQE automatically makes the decision based on its own analysis to create the advised indexes.

The Db2 for i performance tools classify these autonomic indexes as a Maintained Temporary Index (MTI). The Index Advisor output in Figure 19 shows a reference to the MTI term. When the SQE query optimizer decides to automatically create an index based on the index advice, the Index Advisor repository is updated to reflect that fact. The three columns of MTI information for each advised index make it very easy for you to determine which index advice has been turned into an autonomic index by SQE.



Database: Tplxe5 Advised Indexes for Tplxe5

Most Expensive	Average of Query Estimates	Rows in Table when Advised	NLSS Table Advised	NLSS Schema Advised	MTI Used	MTI Created	MTI Last Used	EVI Distinct Values
16	.0000	11718	*HEX		0	0		0
16	.0000	20	*HEX		4	3	12/12/07 3:53:3...	0
16	.0000	10	*HEX		4	3	12/12/07 3:53:3...	0
1	.0000	12	*HEX		0	0		0
1	.0000	5	*HEX		0	0		65537
1	.0000	5	*HEX		0	0		0
1	.0000	3	*HEX		0	0		0
1	.0000	7	*HEX		0	0		0
1	.0000	7	*HEX		0	0		0
1	.0000	0	*HEX		0	0		0
1	.0000	6	*HEX		0	0		0
1	.0000	6	*HEX		0	0		0
1	.0000	6	*HEX		0	0		0
1	.0000	6	*HEX		0	0		0
1	.0000	3	*HEX		0	0		0

28 - 42 of 86 objects

Figure 19 - System-wide index advisor MTI example

The maintenance aspect of SQE temporary indexes is a significant differentiator from the temporary indexes that are created by the CQE query optimizer. This attribute enables SQE temporary indexes to be reused and shared across different queries and jobs. CQE temporary indexes are not maintained so they could not be reused and shared.

While SQE's creation of autonomic indexes can improve performance, the fact that these indexes are temporary objects can also create interesting performance experiences. When a system is switched off, all of the autonomic indexes are deleted because they are temporary system objects. As soon as the system is up and running again, the performance for any SQL statements that were relying on the autonomic index, before the system was restarted, will be slower. This is due to the fact that the query optimizer usually requires several executions of a query in order to learn from the repeated index advice before it can justify the cost of creating the autonomic index. When the optimizer sees that a query runs frequently and the benefits of using the index outweigh the costs of creating it, the optimizer is likely to create the advised index. In other words, the justification is based on the fact that the one-time cost of creating the MTI will be more than offset by the runtime savings in subsequent query runs of this query and others. Autonomic indexes will also be dropped from the system if all of the SQL statements that rely on the MTI are removed from the SQE Plan Cache.

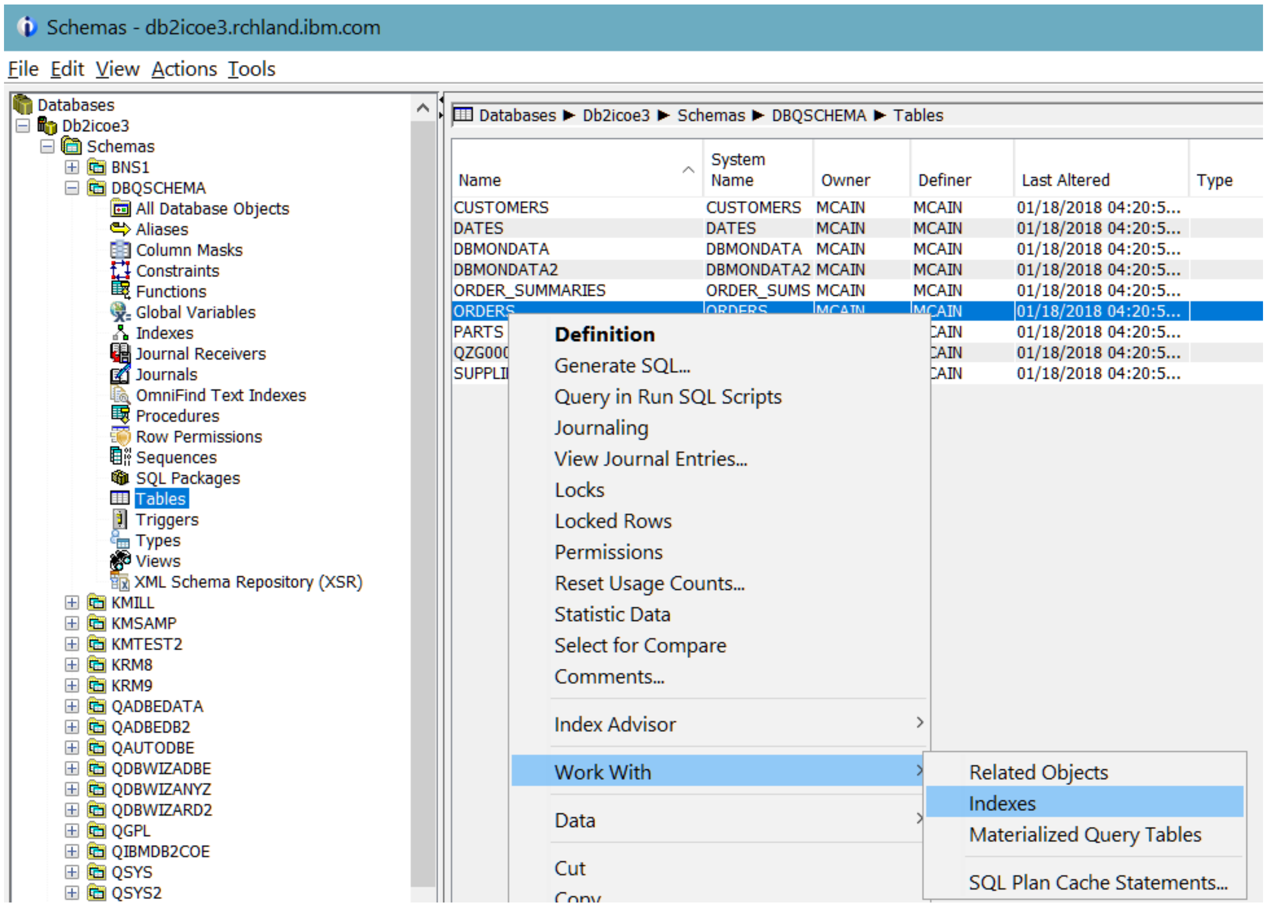
To avoid the experience of having SQL requests performing slower after a system restarts, you should strongly consider creating a permanent index to replace the temporary autonomic indexes. The index advisor output can be sorted by any of the columns, so it is very to easy to use one of the MTI columns to get a list of the advised indexes that have been turned into an autonomic index. Note that the default report layout places the MTI columns on the far right side of the Index Advisor output window (you must scroll right to find these columns).

Starting with the Db2 for i 7.3 release, you can retrieve all the details for the MTI objects currently on your system using the MTI_INFO service.

Index Evaluator

As indexes are created as part of the performance tuning process, you often want to know which indexes are providing benefit to the query optimizer and Db2 engine and the indexes that are not providing value. This index usage analysis is easily accomplished with the Index Evaluator tool. The Index Evaluator provides usage information on all of the indexes that are available for the query optimizer to use. The list of indexes includes those index objects created with SQL along with the indexes supporting primary key constraints, unique constraints, foreign key constraints, keyed logical files, and keyed physical file.

The Index Evaluator is launched by right-clicking a table object and selecting the **Work With → Indexes** task, as shown in Figure 20.

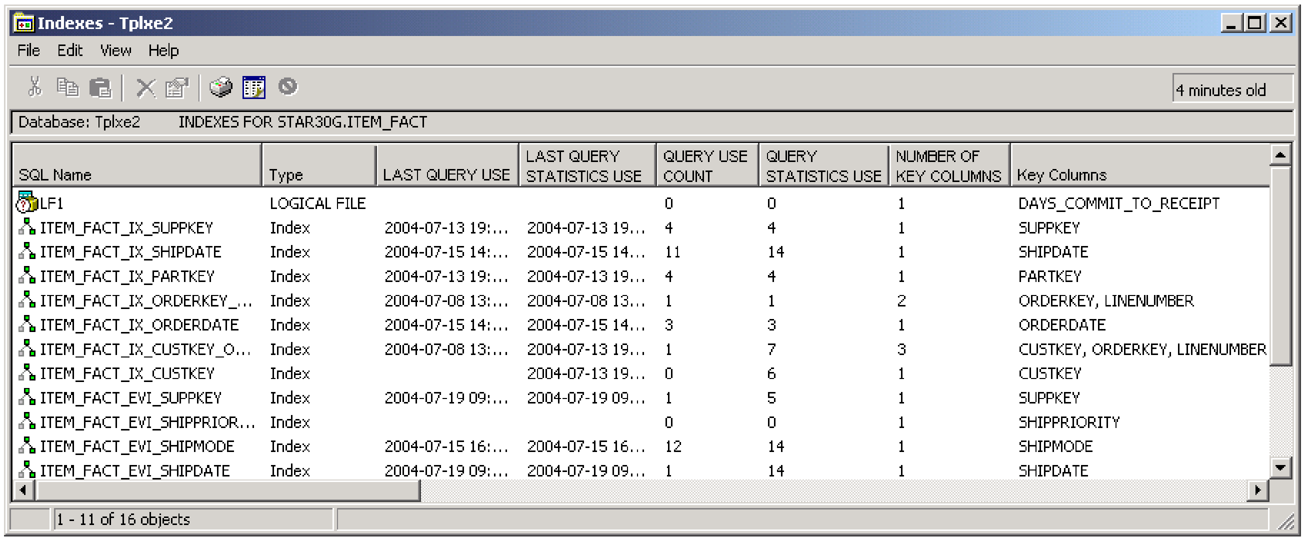


Name	System Name	Owner	Definer	Last Altered	Type
CUSTOMERS	CUSTOMERS	MCAIN	MCAIN	01/18/2018 04:20:5...	
DATES	DATES	MCAIN	MCAIN	01/18/2018 04:20:5...	
DBMONDATA	DBMONDATA	MCAIN	MCAIN	01/18/2018 04:20:5...	
DBMONDATA2	DBMONDATA2	MCAIN	MCAIN	01/18/2018 04:20:5...	
ORDER_SUMMARIES	ORDER_SUMS	MCAIN	MCAIN	01/18/2018 04:20:5...	
ORDERS	ORDERS	MCAIN	MCAIN	01/18/2018 04:20:5...	
PARTS			CAIN	01/18/2018 04:20:5...	
QZG000			CAIN	01/18/2018 04:20:5...	
SUPPLI			CAIN	01/18/2018 04:20:5...	

Figure 20 - Index Evaluator interface

As soon as this task has been selected, ACS returns the evaluation results for all of the indexes defined over the specified table in a new window (see Figure 21). You might be thinking that you can just review the **Last Used Date** object attribute with the DSPFD (Display File Description) command to at least determine the last time the optimizer used an index, but that method will not work. The query optimizer does not always update the **Last Used Date** attribute when using indexes for statistics or query execution purposes.

The Index Evaluator output window in Figure 21 has been customized to move the query usage columns to the front of the window. Normally, a user must scroll to the right to access the query usage columns for an index. The first two columns contain the index name and type of index (SQL index, keyed logical file, and so on). The next four columns contain the data that will help you determine the value of an index from a query optimizer perspective. The **LAST QUERY USE** and **LAST QUERY STATISTICS USE** contain the timestamp value of the last time an index object was used by the optimizer for statistics or for running a query. Correspondingly, the **QUERY USE COUNT** and **QUERY STATISTICS USE COUNT** columns tell you how often the index was used for one of these purposes.



Database: Tplxe2   INDEXES FOR STAR30G.ITEM_FACT

SQL Name	Type	LAST QUERY USE	LAST QUERY STATISTICS USE	QUERY USE COUNT	QUERY STATISTICS USE	NUMBER OF KEY COLUMNS	Key Columns
LF1	LOGICAL FILE			0	0	1	DAYS_COMMIT_TO_RECEIPT
ITEM_FACT_IX_SUPPKEY	Index	2004-07-13 19:...	2004-07-13 19...	4	4	1	SUPPKEY
ITEM_FACT_IX_SHIPDATE	Index	2004-07-15 14:...	2004-07-15 14...	11	14	1	SHIPDATE
ITEM_FACT_IX_PARTKEY	Index	2004-07-13 19:...	2004-07-13 19...	4	4	1	PARTKEY
ITEM_FACT_IX_ORDERKEY_...	Index	2004-07-08 13:...	2004-07-08 13...	1	1	2	ORDERKEY, LINENUMBER
ITEM_FACT_IX_ORDERDATE	Index	2004-07-15 14:...	2004-07-15 14...	3	3	1	ORDERDATE
ITEM_FACT_IX_CUSTKEY_O...	Index	2004-07-08 13:...	2004-07-13 19...	1	7	3	CUSTKEY, ORDERKEY, LINENUMBER
ITEM_FACT_IX_CUSTKEY	Index		2004-07-13 19...	0	6	1	CUSTKEY
ITEM_FACT_EVI_SUPPKEY	Index	2004-07-19 09:...	2004-07-19 09...	1	5	1	SUPPKEY
ITEM_FACT_EVI_SHIPPRIOR...	Index			0	0	1	SHIPPRIORITY
ITEM_FACT_EVI_SHIPMODE	Index	2004-07-15 16:...	2004-07-15 16...	12	14	1	SHIPMODE
ITEM_FACT_EVI_SHIPDATE	Index	2004-07-19 09:...	2004-07-19 09...	1	14	1	SHIPDATE

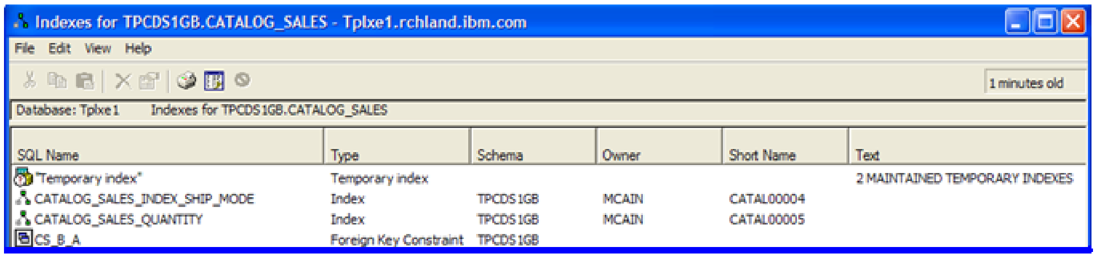
1 - 11 of 16 objects

Figure 21 - Index Evaluator output

Before you start deleting all the indexes and logical files with low usage counts or old timestamp values, you need to first review this information in perspective with how frequently some of your reports and applications are run. For example, assume that there is an index that is only used to speed up the performance of a long running report that is only run as part of the fiscal year-end processing. The usage metrics for this index associated with year-end processing will likely be much lower than other indexes. If you have decided to delete low usage indexes after deeper analysis, it is a good idea to document the definition of the index being deleted in case you later run into a performance problem caused by the deleted index. An easy way to document an index definition is by using the Generate SQL task within ACS.

After creating new indexes for a table, you may want to consider resetting the usage counts back to 0 for all the existing indexes for a table. Resetting the counts enables you to see how often different indexes are being used now that a new set of indexes is in place for the query optimizer to use. The counts can be reset back to 0 by right-clicking on an object within ACS and clicking the **Reset Usage Counts** task.

As shown in Figure 22, the Index Evaluator only returns summary information when a Db2 table has temporary autonomic indexes created over it. The **Text** column contains a count on the number of MTIs currently associated with the table. The easiest way to find the key definitions for the MTI associated with the table is to utilize the Index Advisor task for that same table.



Indexes for TPCDS1GB.CATALOG_SALES - Tplxe1.rchland.ibm.com

File Edit View Help

1 minutes old

Database: Tplxe1 Indexes for TPCDS1GB.CATALOG_SALES

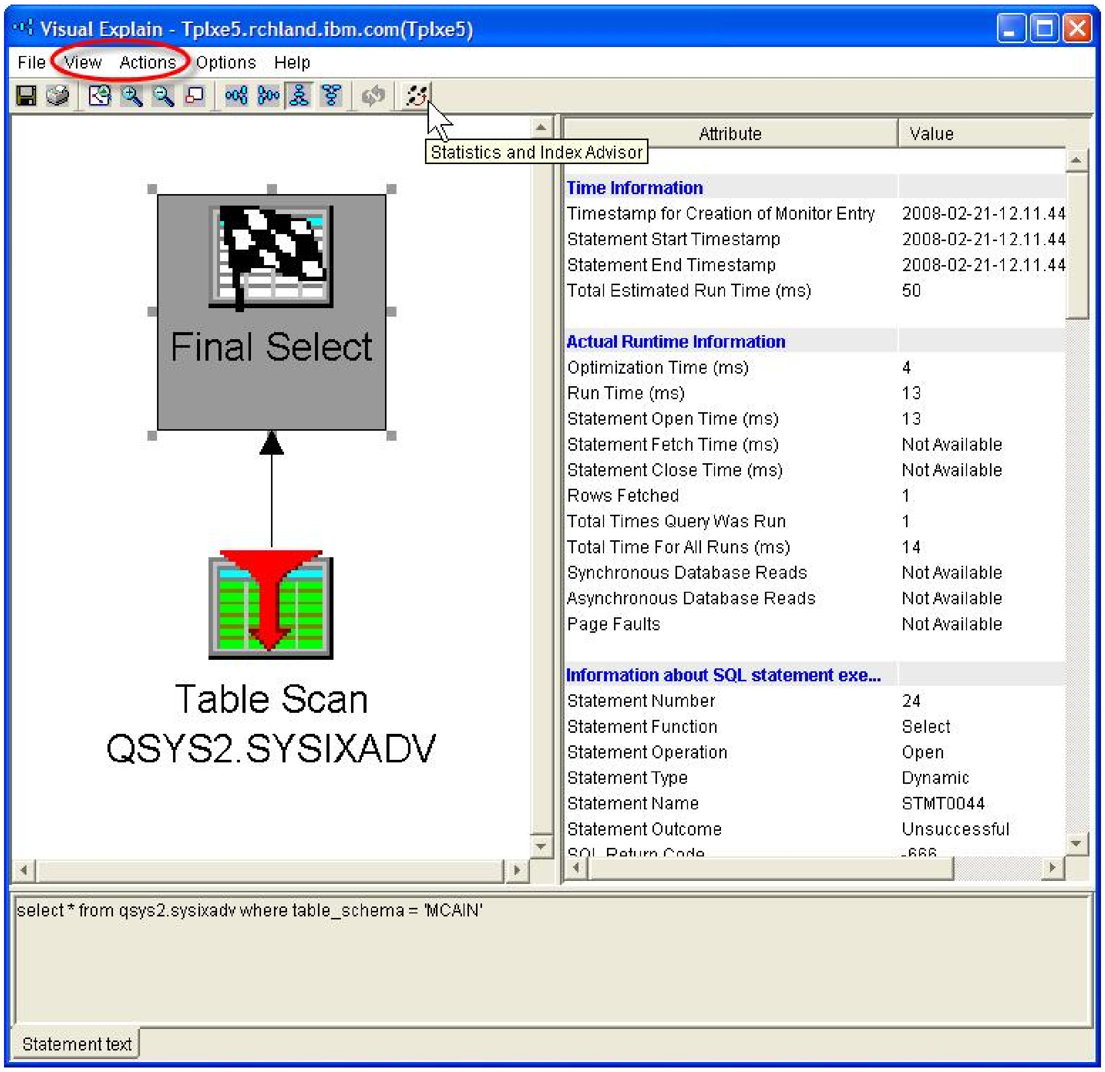
SQL Name	Type	Schema	Owner	Short Name	Text
"Temporary index"	Temporary index				2 MAINTAINED TEMPORARY INDEXES
CATALOG_SALES_INDEX_SHIP_MODE	Index	TPCDS1GB	MCAIN	CATAL00004	
CATALOG_SALES_QUANTITY	Index	TPCDS1GB	MCAIN	CATAL00005	
CS_B_A	Foreign Key Constraint	TPCDS1GB			

Figure 22 - Index Evaluator output for MTIs

The Index Evaluator's query usage metrics can be accessed programmatically by querying one of the following catalog views in QSYS2: SYSINDEXSTAT, SYSPARTITIONINDEXES, or SYSTABLEINDEXSTAT. The Db2 for i SQL Reference contains additional documentation on these catalog views. All of the Db2 for i reference documents can be found online at: **ibm.biz**/db2iBooks. As mentioned earlier, the DSPFD command does not return the query usage metrics for indexes or keyed logical files.

Visual Explain

Visual Explain is the tool that you need to be using to understand and analyze query implementation plans and optimizer feedback. A picture is worth a thousand words and Visual Explain heavily utilizes graphics to simplify the analysis of query access plans. Like the other Db2 for i performance tools, Visual Explain is available through the ACS client. Figure 23 demonstrates how Visual Explain uses a combination of diagrams and text to display the query access plan along with the environment details about the system, job, and application running the SQL request.



Attribute	Value
Time Information	
Timestamp for Creation of Monitor Entry	2008-02-21-12.11.44
Statement Start Timestamp	2008-02-21-12.11.44
Statement End Timestamp	2008-02-21-12.11.44
Total Estimated Run Time (ms)	50
Actual Runtime Information	
Optimization Time (ms)	4
Run Time (ms)	13
Statement Open Time (ms)	13
Statement Fetch Time (ms)	Not Available
Statement Close Time (ms)	Not Available
Rows Fetched	1
Total Times Query Was Run	1
Total Time For All Runs (ms)	14
Synchronous Database Reads	Not Available
Asynchronous Database Reads	Not Available
Page Faults	Not Available
Information about SQL statement exe...	
Statement Number	24
Statement Function	Select
Statement Operation	Open
Statement Type	Dynamic
Statement Name	STMT0044
Statement Outcome	Unsuccessful
SQL Return Code	-666

```
select * from qsys2.sysixadv where table_schema = 'MCAIN'
```

Figure 23 - Visual Explain tool

Visual Explain output can only be produced from a detailed Database Monitor collection, the live SQE Plan Cache, or an SQE Plan Cache Snapshot, or when using the Run SQL Scripts interface. When using the Run Scripts interface, the Explain menu can be used to dynamically generate Visual Explain output for an SQL statement. This interface automatically starts and stops a detailed database monitor in the background to enable the Visual Explain output to be displayed for the SQL statement being run or explained.

The Visual Explain output window also provides access to Index Advisor feedback. The index advice can be accessed in several different ways. The simplest option is to click on the footprints icon on the toolbar at the top of the Visual Explain output window. (This icon is highlighted by the cursor arrow in Figure 23.) The Action pull-down menu also allows you to bring up the Index Advisor. The Highlight pull-down menu provides an indirect method for accessing index advice data. This Highlight menu provides a **Highlight Index Advised** option that is used to emphasize the nodes with a different color in the graphical query plan to make it easy to find those parts of the query plan that are associated with the generated index advice.

Temporary index feedback

When the query plan displayed by Visual Explain utilizes temporary indexes, there is a good chance that the CQE Index Advisor will recommend indexes that are different from the temporary indexes shown in the query plan. When processing query plans built by CQE, Visual Explain tries to enhance the CQE basic index advice that is focused on local selection predicates by also analyzing the usage of temporary indexes within the query plan. Combining these two pieces of feedback enables Visual Explain to provide index advice that is closer to a perfect index. Consider a query that joins two tables, where the first table is accessed with a full table scan to return a subset of the rows and the second table is accessed with a temporary index. The CQE Index Advisor recommends an index based only on the local selection on the first table. The Visual Explain Index Advisor interface also suggests an index for the join column(s) of the second table based on the fact that the query plan produced by the CQE optimizer contains a temporary index built for nested loop join processing. Consider using all your knowledge of the data and query to build an index over any columns recommended by the CQE Index Advisor plus key columns in the temporary index used for the nested loop join. While the Visual Explain interface tries to enhance the CQE index advice by analyzing other portions of the query, the Visual Explain advice is still limited when compared to the SQE Index Advisor.

As these recommendations may yield several indexes, all designed for the same query. Best practices dictate running the query again with all of the recommended indexes present. If the desired results are still not being achieved, consider creating a radix index that combines all of the columns in the following priority: selection predicates with equalities, join predicates, and then one selection predicate defined with inequalities. If the WHERE clause contains only join predicates, ensure that a radix index exists over each table in the join. The join column(s) must be in the primary or left-most position of the key.

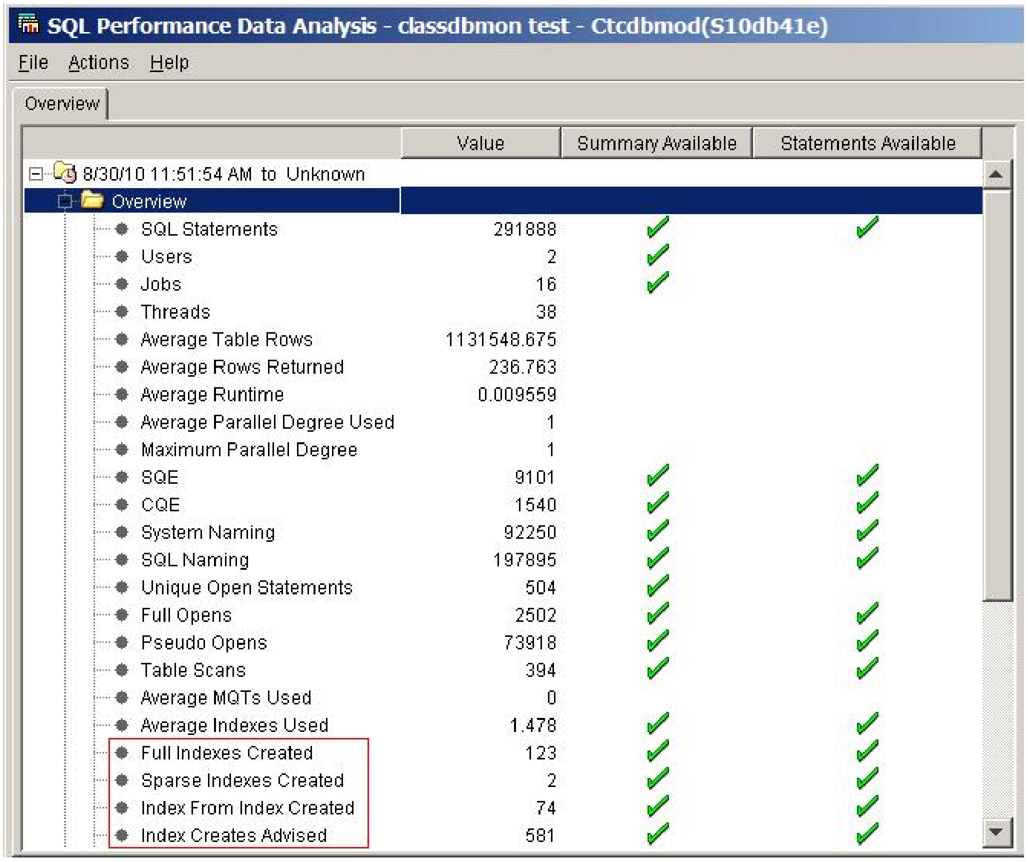
SQL Performance Monitor (Database Monitor)

With its base installation, Db2 for i includes an SQL Performance Monitors that provides the ability to collect either detailed monitor data or summary-level monitor data. As highlighted in Figure 12, only the detailed database monitor function should be used because the summary monitor is nonstrategic and no longer being enhanced by IBM.

The detailed database monitor is a trace tool that collects low-level Db2 details while SQL statements are running on the system. The detailed monitor also collects a subset of information for queries originating from the Open Query File (OPNQRYF) command and Query/400 product, allowing Visual Explain to support these legacy query interfaces as well. The collected data is written to a user-specified output table. After the monitor data has been collected, Visual Explain or reports can be run against the collected data in the output table to analyze and understand the optimization and runtime performance of the SQL request. This analysis helps to identify and tune Db2 performance problem areas.

The detailed database monitor can be started and stopped either using the SQL performance monitor interface within ACS or with the STRDBMON (Start Database Monitor) and ENDDBMON (End Database Monitor) system commands. Connection properties that automatically collect detailed monitor data for a connection are available on the ODBC, JDBC, and ADO.NET middleware that is shipped with the IBM i ACS product. Due to the overhead generated by the detailed monitor, it is best to try and run the monitor for short durations of time and target a limited number of jobs or connections.

In terms of index analysis and tuning, a database monitor collection will contain details on the indexes used in a query implementation, whether they are permanent or temporary indexes, along with any generated index advice. An example of the database monitor's support for index analysis is highlighted for your reference in Figure 24. This summarization of the data collected by the database monitor was produced by the SQL Performance Monitor Analyze task within ACS.


SQL Performance Data Analysis - classdbmon test - Ctcdbmod(S10db41e)
File
Actions
Help
Overview

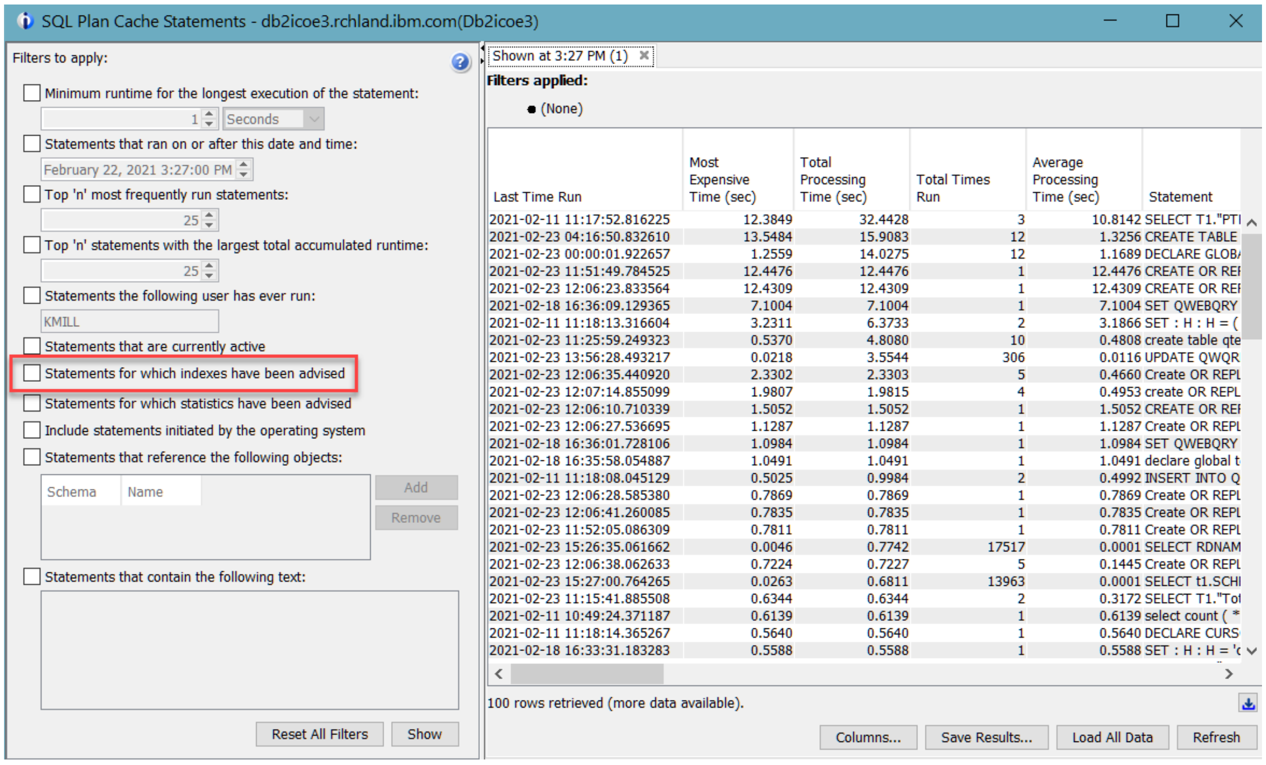
	Value	Summary Available	Statements Available
8/30/10 11:51:54 AM to Unknown			
Overview			
SQL Statements	291888	✓	✓
Users	2	✓	
Jobs	16	✓	
Threads	38		
Average Table Rows	1131548.675		
Average Rows Returned	236.763		
Average Runtime	0.009559		
Average Parallel Degree Used	1		
Maximum Parallel Degree	1		
SQE	9101	✓	✓
CQE	1540	✓	✓
System Naming	92250	✓	✓
SQL Naming	197895	✓	✓
Unique Open Statements	504	✓	
Full Opens	2502	✓	✓
Pseudo Opens	73918	✓	✓
Table Scans	394	✓	✓
Average MQTs Used	0		
Average Indexes Used	1.478	✓	✓
Full Indexes Created	123	✓	✓
Sparse Indexes Created	2	✓	✓
Index From Index Created	74	✓	✓
Index Creates Advised	581	✓	✓

Figure 24 - Database Monitor summary output example

It is often beneficial to use the Database monitor data in conjunction with the System-wide Index Advisor to identify the SQL statements and queries that are causing the index advice to be generated. The system-wide index advisor tool can only link the index advice to SQL statements that have plans currently stored in the SQE Plan Cache. Because SQL statements processed by CQE do not have plans stored in the SQE Plan Cache, the database monitor tool would be required for finding the SQL statements associated with CQE index advice.

SQE Plan Cache

A graphical interface for the SQL Query Engine Plan Cache was added to the performance toolset in V5R4. The Plan Cache enables you to get a view of all the SQL statements that have been running on a system. This tool can be used several times a day to check on the top 10 longest running SQL statements or the most frequently executed statements. With these filters, you can focus your detailed analysis on those statements that will have the biggest impact on your overall Db2 performance.



SQL Plan Cache Statements - db2icoe3.rchland.ibm.com(Db2icoe3)

Filters to apply:

- [ ] Minimum runtime for the longest execution of the statement: 1 Seconds
- [ ] Statements that ran on or after this date and time: February 22, 2021 3:27:00 PM
- [ ] Top 'n' most frequently run statements: 25
- [ ] Top 'n' statements with the largest total accumulated runtime: 25
- [ ] Statements the following user has ever run: KMILL
- [ ] Statements that are currently active
- [ ] Statements for which indexes have been advised
- [ ] Statements for which statistics have been advised
- [ ] Include statements initiated by the operating system
- [ ] Statements that reference the following objects: Schema Name
- [ ] Statements that contain the following text:

Shown at 3:27 PM (1)

Filters applied:

- (None)

Last Time Run	Most Expensive Time (sec)	Total Processing Time (sec)	Total Times Run	Average Processing Time (sec)	Statement
2021-02-11 11:17:52.816225	12.3849	32.4428	3	10.8142	SELECT T1."PTI
2021-02-23 04:16:50.832610	13.5484	15.9083	12	1.3256	CREATE TABLE
2021-02-23 00:00:01.922657	1.2559	14.0275	12	1.1689	DECLARE GLOB/
2021-02-23 11:51:49.784525	12.4476	12.4476	1	12.4476	CREATE OR REI
2021-02-23 12:06:23.833564	12.4309	12.4309	1	12.4309	CREATE OR REI
2021-02-18 16:36:09.129365	7.1004	7.1004	1	7.1004	SET QWEBQRY
2021-02-11 11:18:13.316604	3.2311	6.3733	2	3.1866	SET : H : H = (
2021-02-23 11:25:59.249323	0.5370	4.8080	10	0.4808	create table qte
2021-02-23 13:56:28.493217	0.0218	3.5544	306	0.0116	UPDATE QWQR
2021-02-23 12:06:35.440920	2.3302	2.3303	5	0.4660	Create OR REPL
2021-02-23 12:07:14.855099	1.9807	1.9815	4	0.4953	create OR REPL
2021-02-23 12:06:10.710339	1.5052	1.5052	1	1.5052	CREATE OR REI
2021-02-23 12:06:27.536695	1.1287	1.1287	1	1.1287	Create OR REPL
2021-02-18 16:36:01.728106	1.0984	1.0984	1	1.0984	SET QWEBQRY
2021-02-18 16:35:58.054887	1.0491	1.0491	1	1.0491	declare global t
2021-02-11 11:18:08.045129	0.5025	0.9984	2	0.4992	INSERT INTO Q
2021-02-23 12:06:28.585380	0.7869	0.7869	1	0.7869	Create OR REPL
2021-02-23 12:06:41.260085	0.7835	0.7835	1	0.7835	Create OR REPL
2021-02-23 11:52:05.086309	0.7811	0.7811	1	0.7811	Create OR REPL
2021-02-23 15:26:35.061662	0.0046	0.7742	17517	0.0001	SELECT RDNAM
2021-02-23 12:06:38.062633	0.7224	0.7227	5	0.1445	Create OR REPL
2021-02-23 15:27:00.764265	0.0263	0.6811	13963	0.0001	SELECT t1.SCHI
2021-02-23 11:15:41.885508	0.6344	0.6344	2	0.3172	SELECT T1."Tot
2021-02-11 10:49:24.371187	0.6139	0.6139	1	0.6139	select count (*
2021-02-11 11:18:14.365267	0.5640	0.5640	1	0.5640	DECLARE CURS
2021-02-18 16:33:31.183283	0.5588	0.5588	1	0.5588	SET : H : H = 'c

100 rows retrieved (more data available).

Figure 25 - SQL Plan Cache interface

Using the SQL Plan Cache tool is as simple as clicking the **SQL Performance Center** link on the main ACS interface. Once, the SQL Performance Center window is displayed – click on the **Show Statements** button. This action displays the interface as shown in Figure 25. When the **Apply** button is clicked, the interface will be populated with the SQL statements that meet the filter criteria. From there, performance analysis can be done using Visual Explain by right-clicking an SQL statement and selecting the Visual Explain task.

On the left-hand side, notice that there is a filter that enables you to only view those statements associated with advised indexes. This filter makes it easy to quickly focus your analysis to those SQL statements that have index recommendations.

Before you get too carried away with this exciting new tool, remember that the SQL Plan Cache is only populated for SQL statements processed by SQE. Statements processed by CQE do not have entries in the Plan Cache.

Plan Cache Snapshots

Db2 also provides a way to dump the contents of the Plan Cache into a permanent table known as a snapshot. Dumping the plan data into a Plan Cache Snapshot, allows performance analysis to be done on static data, eliminating the need to worry about some of the query plans being removed from the Plan Cache due to size limitations. The Snapshot functionality can also be used to capture Plan Cache information on a regular basis. Capturing snapshots on a regular basis enables you to perform before and after comparisons to better understand the reason for a change in SQL performance on a system.

Additional tools

The SYSTOOLS schema contains two stored procedures, HARVEST_INDEX_ADVICE and ACT_ON_INDEX_ADVICE, to make it easier for users to analyze and utilize the data contained in the system-wide index advisor repository. The HARVEST_INDEX_ADVICE procedure generates an SQL script of Create Index statements based on the data in the system-wide index advisor repository. This generated script can be modified and run to create the appropriate indexes. The ACT_ON_INDEX_ADVICE procedure uses the same index advisor repository, but actually creates the index instead of generating an SQL script.

Any future SYSTOOL utilities that IBM delivers to assist with indexing your database will be highlighted into the Db2 for i Technology Updates wiki at: **ibm.com**/ibmi/techupdates/db2

Summary

As with all relational database management systems, indexes and statistics play an important role in query optimization and query execution. The presence of the correct number and type of indexes is a critical success factor in achieving excellent SQL performance. In addition to assisting in data retrieval, indexes also provide valuable information and statistics to the Db2 for i cost-based query optimizer. The ability to provide the optimizer with information about selectivity, data skew, and column cardinality becomes critically important as databases scale into the hundreds of gigabytes and eventually terabytes of data. Advanced techniques such as encoded vector indexes and the ability to use the maintained aggregates contained within the index are powerful advantages.

Knowing more and doing more with Db2 for i can help you deliver business requirements with a very low total cost of ownership.

Appendix A - Resources

The following websites provide useful references to supplement the information contained in this paper:

- IBM i Access Client Solutions (ACS)
  ibm.biz/IBMi_ACS

- Db2 for i online manuals
  ibm.biz/db2iBooks

- Database Engineer (DBE) Information and Skills Transfer
  ibm.biz/Db2iExpertLabs

- IBM Redbooks - Db2 for i
  ibm.biz/db2iRedbooks

- Db2 for i white papers
  ibm.biz/db2iPapers

- Db2 for i Blog
  Db2IBMi.blogspot.com

Appendix B - Example Queries & Indexing Strategies

The following are some examples of SQL query requests with a recommended set of indexes to create. The purpose of these examples is to demonstrate the concept of proactive index creation based on the actual SQL request and knowledge of the query optimizer and database engine. These examples are only listed to illustrate one proactive methodology. The actual implementation and performance of these SQL requests is dependent upon several factors, including, but not limited to: database table and index sizes, version of IBM i and Db2 for i, query interface attributes, job and system attributes, and environment. The query plans and performance results might vary.

Example 1

```
SELECT *
      FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR)
```

Or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
```

Anticipating index probe or table probe with dynamic bitmap

Example 2

```
SELECT *
      FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE)
```

Keys can be in any order, most selective column first

Or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
```

Anticipating index probe or table probe with dynamic bitmaps

Example 3

```
SELECT *
      FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')
      AND A.STYLE = 'ADULT MENS T-SHIRT'

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE, STYLE)
```

Keys can be in any order, most selective columns first

Or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI3 ON TABLE1 (STYLE)
```

Anticipating index probe or table probe with dynamic bitmaps

Example 4

```
SELECT * FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')
      AND A.STYLE = 'ADULT MENS T-SHIRT'
      AND A.INVENTORY > 100

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE, STYLE, INVENTORY)
```

Keys can be in any order, most selective columns first, nonequal predicate last

Or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI3 ON TABLE1 (STYLE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI4 ON TABLE1 (INVENTORY)
```

Anticipating index probe or table probe with dynamic bitmaps

Example 5

```
SELECT * FROM TABLE1 A, TABLE2 B
      WHERE A.KEY = B.KEY
      AND A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')
      AND A.STYLE = 'ADULT MENS T-SHIRT'
      AND A.INVENTORY > 100

CREATE INDEX TABLE1_INDEX1 ON TABLE1
      (COLOR, SIZE, STYLE, KEY, INVENTORY)
```

Keys can be in any order, most selective local selection columns first, nonequal predicate last

```
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (KEY)
```

And / or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI3 ON TABLE1 (STYLE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI4 ON TABLE1 (INVENTORY)
```

Anticipating index probe, or table probe with dynamic bitmaps, and nested loop join with index

Example 6

```
SELECT *
      FROM TABLE1 A,
            TABLE2 B
      WHERE A.KEY_COL1 = B.KEY_COL2
      AND A.COLOR IN ('BLUE', 'GREEN', 'RED')

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, KEY_COL1)
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (KEY_COL2)
```

Anticipating index probe and nested loop join with index

Example 7

```
SELECT A.STORE, A.STYLE, A.SIZE, A.COLOR SUM(A.QUANTITY_SOLD)
      FROM TABLE1 A, TABLE2 B
      WHERE A.KEY = B.KEY
      AND A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')
      AND A.STYLE = 'ADULT MENS T-SHIRT'
      GROUP BY A.STORE, A.STYLE, A.SIZE, A.COLOR

CREATE INDEX TABLE1_INDEX1 ON TABLE1
      (COLOR, SIZE, STYLE, KEY)
```

Keys can be in any order, most selective local selection columns first

```
CREATE INDEX TABLE1_INDEX2 ON TABLE1
      (STORE, STYLE, SIZE, COLOR)
```

Keys must be in this order for grouping stats

```
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (KEY)
```

And / or

```
CREATE ENCODED VECTOR INDEX TABLE1_EVI1 ON TABLE1 (COLOR)
CREATE ENCODED VECTOR INDEX TABLE1_EVI2 ON TABLE1 (SIZE)
CREATE ENCODED VECTOR INDEX TABLE1_EVI3 ON TABLE1 (STYLE)
```

Anticipating index probe, or table probe with dynamic bitmaps, and nested loop join with index

Example 8

```
SELECT A.COL3, A.COL4, B.COL2, C.COL6, C.COL7
      FROM TABLE1 A,
           TABLE2 B,
           TABLE3 C
      WHERE A.KEY_COL1 = B.KEY_COL1
      AND A.KEY_COL2 = C.KEY_COL2
      AND A.COLOR IN ('BLUE', 'GREEN', 'RED')

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, KEY_COL1)
CREATE INDEX TABLE1_INDEX2 ON TABLE1 (COLOR, KEY_COL2)
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (KEY_COL1)
CREATE INDEX TABLE3_INDEX1 ON TABLE3 (KEY_COL2)
```

Anticipating index probe and nested loop join with index

Example 9

```
SELECT A.COLOR, A.SIZE, SUM(A.SALES), SUM(A.QUANTITY)
      FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
      GROUP BY A.COLOR, A.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE)
```

Anticipating index probe and grouping with index

Example 10

```
SELECT A.COLOR, A.SIZE, SUM(A.SALES), SUM(A.QUANTITY)
      FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
      GROUP BY A.COLOR, A.SIZE
      ORDER BY A.COLOR, A.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE)
```

Anticipating index probe, grouping with index, and ordering with index

Example 11

```
SELECT A.COLOR, A.SIZE, MIN(A.QUANTITY)
      FROM TABLE1 A
      WHERE A.COLOR IN ('BLUE', 'GREEN', 'RED')
      GROUP BY A.COLOR, A.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE, QUANTITY)
```

Anticipating index probe and grouping with index with MIN skip processing

Example 12

```
SELECT B.COLOR, B.SIZE, SUM(A.SALES), SUM(A.QUANTITY)
      FROM TABLE1 A,
            TABLE2 B
      WHERE A.KEY_COL1 = B.KEY_COL1
      AND B.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND B.SIZE IN ('LARGE', 'X-LARGE')
      GROUP BY B.COLOR, B.SIZE
      ORDER BY B.COLOR, B.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (KEY_COL1)
CREATE INDEX TABLE2_INDEX1 ON TABLE2 (COLOR, SIZE, KEY_COL1)
```

Anticipating index probe, nested loop join with index, and grouping and ordering with index

Example 13

```
SELECT A.COLOR, A.SIZE, A.SALES
      FROM TABLE1 A
      WHERE A.SALES <  (SELECT AVG(B.SALES)
                        FROM TABLE1 B
                        WHERE B.SIZE IN ('LARGE', 'X-LARGE'))
      AND A.COLOR IN ('BLUE', 'GREEN', 'RED')
      AND A.SIZE IN ('LARGE', 'X-LARGE')

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (SIZE, COLOR)
```

Anticipating index probe and grouping with index. Because SIZE is the first key column in the index, the index can be used for both the inner and outer queries.

Example 14

```
SELECT A.COLOR, A.SIZE, SUM(A.SALES), SUM(A.QUANTITY)
      FROM TABLE1 A
      GROUP BY B.COLOR, B.SIZE

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE)
```

Anticipating grouping with index

Or

```
CREATE INDEX TABLE1_INDEX1 ON TABLE1 (COLOR, SIZE, SALES, QUANTITY)
```

Anticipating grouping with index and index only access (all reference columns in the index)

Example 15

```
SELECT A.CUSTOMER, A.CUSTOMER_NUMBER, A.YEAR, A.MONTH, A.SALES
      FROM TABLE1 A
      WHERE A.SALES > (SELECT AVG(B.SALES)
                           FROM TABLE1 B
                           WHERE B.CUSTOMER = A.CUSTOMER
                           AND B.YEAR = A.YEAR)
      AND A.YEAR = 2001

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (YEAR, CUSTOMER, SALES)
```

Anticipating index probe, index only access (subquery) and grouping with index (subquery) — index used for both queries

Example 16

```
SELECT A.CUSTOMER, A.CUSTOMER_NUMBER, A.YEAR, A.MONTH, A.SALES
      FROM TABLE1 A
      WHERE A.CUSTOMER_NUMBER IN
            (SELECT B.CUSTOMER_NUMBER
                              FROM TABLE2 B
                              WHERE NUMBER_OF_RETURNS > 10
                              AND B.YEAR = 2000
                              AND B.MONTH IN (10, 11, 12))

      AND A.YEAR = 2001
      AND A.MONTH IN (1, 2, 3)

CREATE INDEX TABLE1_INDEX1 ON TABLE1
      (YEAR, MONTH, CUSTOMER_NUMBER)

CREATE INDEX TABLE2_INDEX1 ON TABLE2
      (YEAR, MONTH, CUSTOMER_NUMBER, NUMBER_OF_RETURNS)
```

Anticipating index probe, subquery join composite with nested loop, and join with index

Example 17

```
UPDATE TABLE1
      SET COL1 = 125
      SET COL2 = ‘ABC’
      SET COL3 = ‘This is updated’
      WHERE CUSTOMER_NUMBER IN (4537, 7824, 2907)
      AND YEAR = 2001

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (YEAR, CUSTOMER_NUMBER)
```

Anticipating index probe

Example 18

```
DELETE FROM TABLE1
      WHERE ITEM_NUMBER IN (‘23-462’, ‘45-7124’, ‘21-2007’)
      AND QUANTITY = 0

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (ITEM_NUMBER, QUANTITY)
```

Anticipating index probe

Example 19

```
UPDATE TABLE1 A
      SET LAST_YEARS_MAX_SALES =
            (SELECT MAX(SALES)
                             FROM TABLE2 B
                             WHERE B.YEAR = 2000
                             AND B.MONTH = A.MONTH
                             AND B.CUSTOMER = A.CUSTOMER)
      WHERE A.YEAR = 2001

CREATE INDEX TABLE1_INDEX1 ON TABLE1 (YEAR)

CREATE INDEX TABLE2_INDEX1 ON  TABLE2
      (YEAR, MONTH, CUSTOMER, SALES DESC)
```

Anticipating index probe and grouping with index with MAX skip processing

Example 20

```
SELECT     T.CHAR_DATE,
           C.COUNTRY,
           C.CUSTOMER_NAME,
           P.PART_NAME,
           S.SUPPLIER_NAME,
           SUM(F.QUANTITY),
           SUM(F.REVENUE_WO_TAX)
      FROM STARLIB/SALES_FACTS F,
           STARLIB/PART_DIM P,
           STARLIB/TIME_DIM T,
           STARLIB/CUST_DIM C,
           STARLIB/SUPP_DIM S
      WHERE F.PARTKEY = P.PARTKEY
      AND        F.TIMEKEY = T.TIMEKEY
      AND        F.CUSTKEY = C.CUSTKEY
      AND        F.SUPPKEY = S.SUPPKEY
      AND        T.YEAR = 2000
      AND        T.MONTH = 06
      AND        T.DAY = 30
      AND        C.COUNTRY = 'JAPAN'
      AND        P.MFGR = 'Manufacturer#3'
      GROUP BY   T.CHAR_DATE,
                 C.COUNTRY
                 C.CUSTOMER_NAME,
                 P.PART_NAME,
                 S.SUPPLIER_NAME
      ORDER BY   T.CHAR_DATE,
                 C.COUNTRY,
                 C.CUSTOMER_NAME,
                 P.PART_NAME

CREATE INDEX SALES_FACTS_INDEX1 ON SALES_FACTS (PARTKEY)
CREATE INDEX SALES_FACTS_INDEX2 ON SALES_FACTS (TIMEKEY)
CREATE INDEX SALES_FACTS_INDEX3 ON SALES_FACTS (CUSTKEY)
CREATE INDEX SALES_FACTS_INDEX4 ON SALES_FACTS (SUPPKEY)

CREATE INDEX PART_DIM_INDEX1 ON PART_DIM (MFGR, PARTKEY, PART_NAME)
CREATE INDEX TIME_DIM_INDEX1 ON TIME_DIM (YEAR, MONTH, DAY, TIMEKEY)
CREATE INDEX CUST_DIM_INDEX1 ON CUST_DIM (COUNTRY, CUSTKEY)
CREATE INDEX SUPP_DIM_INDEX1 ON SUPP_DIM (SUPPKEY, SUPPLIER_NAME)
```

Anticipating index probe, index only access and nested loop join with index

About the authors

Kent Milligan is a Senior Db2 for i Consultant in IBM Technology Expert Labs. Kent has over 25 years of experience as a Db2 for IBM i consultant and developer working out of the IBM Rochester lab. Prior to re-joining the DB2 for i Expert Labs practice in 2020, Kent spent 5 years working on healthcare solutions powered by IBM Watson technologies. Kent is a sought-after speaker and author on Db2 for i & SQL topics. You can reach Kent at kmill@us.ibm.com or at his blog: Db2IBMi.blogspot.com.

Mike Cain was a Senior Technical Staff Member within the IBM Systems and Technology Group focusing on Db2 for i performance and leading the Db2 for i Center of Excellence. Prior to that position, he worked as an IBM AS/400 Systems Engineer and Technical Consultant. Before joining IBM in 1988, Mike worked as a System/38 programmer and DP Manager for a property and casualty insurance company. Mike has extensive experience working with customers and business partners, and uses this knowledge to influence the solutions, development and support processes. He is located in Rochester, MN.

Trademarks and special notices