Fastcomcorp Research
White Paper

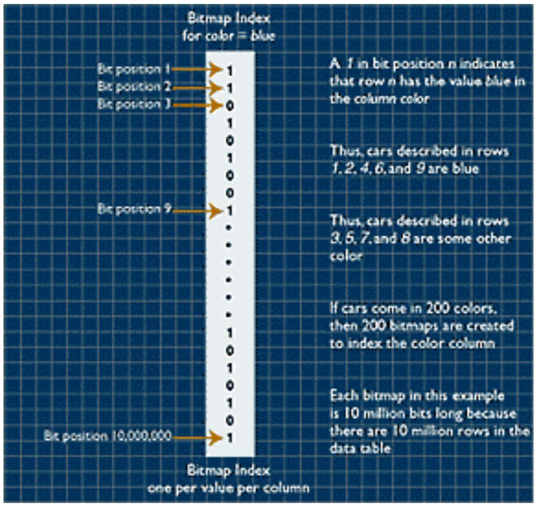
Hyperdex

Introduction by Joseph Richburg

Public Version

Introduction

As SQL tables have gotten larger and demands for business intelligence have increased, the computer industry has pressed for greater performance from its database tools. At the high end, end users seek a higher level of response from applications like data mining and data warehousing. From a bottoms up perspective, systems professionals have a need for basic retrieval technologies that speed up database operations. Both of these needs have focused attention on the performance advantages of bitmap indexing. Over the past few years Oracle with its Bitmap Join and IBM with its Encoded Vector Index have patented bitmap technologies which have produced major gains in the evaluation of complex queries. However, both company's products are restricted in application due to the limitations of their respective bitmap indexing schemes. **Both companies limit bitmap indexing to low cardinality data fields and neither supports bitmap indexing for high cardinality fields such as numerical data fields, range-based data fields or text data fields**.


Bitmap Index
for color = blue
Bit position 1
Bit position 2
Bit position 3
Bit position 9
Bit position 10,000,000
1
1
0
1
0
1
0
0
1
1
0
1
0
1
0
1
A 1 in bit position n indicates that row n has the value blue in the column color
Thus, cars described in rows 1, 2, 4, 6, and 9 are blue
Thus, cars described in rows 3, 5, 7, and 8 are some other color
If cars come in 200 colors, then 200 bitmaps are created to index the color column
Each bitmap in this example is 10 million bits long because there are 10 million rows in the data table
Bitmap Index
one per value per column

Hyperdex is a dramatic new development in bitmap index technology. It creates efficient and highly scalable bitmaps which are capable of indexing any and all data fields in a database table. It can also perform complex queries on any or all such indexed fields with sub-second response times. It's a genuine "gee whiz" technology as can be seen in the performance displays. Hyperdex is a major step in the evolution of database indexing technologies, particularly Bitmap Indexing. At a time when databases are getting much larger and users are requiring broader and faster access to their data, new database structures like fact tables and cubes and new applications like data warehousing, data mining, and business intelligence are fast emerging. Unfortunately, the underlying data access methods have been slow developing.

This current concentration on new business solutions has focused attention on the bitmap where an assignment of value (1 or 0) registers the presence or absence of an attribute value. It has been shown that such a simple translation can greatly improve the access speed to the underlying data. Likewise, it has also been observed that data fields with medium to high cardinality (more than a few distinct values) require an inordinate number of bitmaps to encode their data.

Also, IBM Corporation, has patented an improvement of the basic bitmap called the Encoded Vector Index (EVI). The idea here is to add a step in bitmap creation, the creation of a symbol table (EVI), which is then used to create bitmaps. And even though Bitmap Indexes, Bitmap Join Indexes and EVIs represent the state of the art for today's database access methods, they are far too limiting. What do we do for data fields with higher cardinality; i.e., numerical fields or text data? How do we get beyond simple equality coding to encode for range or interval searches?

Hyperdex Conceptual Model

To gain significant improvement over current bitmap indexing techniques, Hyperdex starts with a basic bitmap and lays a foundation for a more comprehensive approach.
First, any bitmap can be conceptualized as a series of points along a line whose length is the size of the indexed table. Each point along the line corresponds to a row position in the table and its value (1 or 0) corresponds to an attribute value for one of the row's columns. For example, if the bitmap is for the Alaska attribute of the State column, then a 1 in the nth bitmap position codes the corresponding row's State value as Alaska. Other attributes for the State column would have their own separate bitmaps, and additional bitmaps would be necessary to code attributes for other columns. Thus, the conceptual model that is described becomes two dimensional with say the database dimension as the horizontal component and the column attributes the vertical component. While simple in concept this model hints at the structure of the Hyperdex model.

In data warehousing environments the size of bitmap indexes can get very large. For example, a table with 1 million rows requires a minimum of 125 thousand bytes for each bitmap. If the field being indexed has a cardinality of as few as 50 values, the bitmap index would require over 6 megabytes (50 x 125k). Since more typical fact tables can contain as many as 109 rows, such an index would require 1000 times 6 million or more than 6 gigabytes.

The question becomes, "*How does one store and access such a structure?*" Disk storage of large bitmap indexes is not considered that much of a challenge. If necessary, the index could be easily horizontally partitioned and processed in discrete sections. Consideration could also be given to storing each section on a different hard disk to enhance query performance. While storage appears less of a problem, index access presents a greater challenge. Since all bitmaps are congruent on their horizontal dimension, they can be efficiently combined to answer more complex queries. This alone is sufficient motivation to keep the database dimension inviolate. For new indexing techniques, attention must be focused on the vertical dimension, which determines the bits that are set to 1 in each bitmap of the index.

Range of the Interval Encoding

As discussed above separate bitmaps can be used to code low cardinality attributes, but the question arises, "Can this concept be extended to higher cardinality attributes." For example, can numeric attribute values be coded and simply stored vertically in multiple bitmaps. A quick drawback of this scheme is that to process a query, most of the bitmaps would have to be accessed, and possibly multiple times. Also, such a technique would not directly facilitate querying range and inequality predicates.

One of the first barriers *Hyperdex had to face was coding bitmaps for numeric, high cardinality data fields*. Numeric data fields are quite prevalent in all kinds of databases, especially fact tables. Moreover, numeric data queries rarely involve equality testing. They typically involve range testing and inequality operations such as less than (<), greater than (>), less than or equal (<=), greater than or equal (>=), etc.

To accomplish range and interval encoding Hyperdex engineers developed their own form of encoded bitmap indexing. In this technique all numeric values regardless of their native data types are converted to a common binary data type. Field values are detected for their absolute domain, scaled to a common factor, and converted to long integers. This permits bit-wise operations to be used to construct efficient and effective bitmaps for the data field.

However, in line with the idea that it is not efficient to access, but a few bitmaps to resolve a query, a novel magnitude encoding technique was developed to minimize the number of bitmaps accessed. Since this concept already included a sense of magnitude as well as value, it was ideal for both equality and inequality processing.

Having invented a very efficient method for coding numeric data, Hyperdex engineers saw its immediate application to low cardinality data. Just as IBM used a symbol table (EVI) to assign numeric values to a column's attribute values, Hyperdex uses a symbol table to generate a numeric value for each distinct value in a low cardinality data field. It then uses its proprietary numeric algorithm to index the value. Conversely, when querying such a field, the same symbol table is used to get an attribute's numerical equivalent which is used to perform the search.

Text Encoding

Many high cardinality data fields are not numeric, but contain text-like descriptive information. Names, titles, departments, cities, regions, customers, part numbers, descriptions, and other such data fields add an important dimension to databases, and Hyperdex would not be complete without a way to index text. While presence/absence has been shown to work well for low cardinality data fields and magnitude was considered essential for coding numeric fields, some other intrinsic characteristic had to be found for text data.

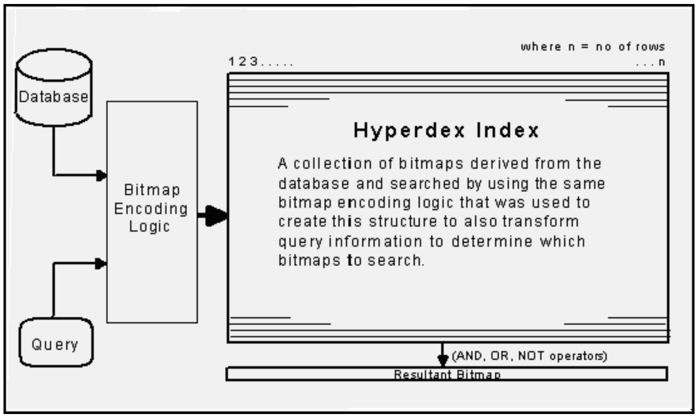
Techniques for indexing text using bitmaps is not so well known, but a search of the literature points to ways this has been accomplished. **The two most common text indexing techniques are inverted files and signature files**. An inverted file contains a list of all the values being indexed along with the identifiers of the records containing the values. In a signature file each record is allocated a fixed-width signature, or bit string, of *w* bits. Each word that is indexed is hashed to determine the bits in the signature to be set. Of these two methods the signature file is closest to bitmap indexing.

Experience has shown that typical full-word indexing techniques are not as efficient as partial-match retrieval using the method of superimposed codes. In this method a smaller number of key values in the text are identified and indexed. The records selected and retrieved are those having a match to all similarly derived key values in a query set. See the article "Partial-Match Retrieval via the Method of Superimposed Codes" Roberts C. S. (1979) Proceedings of IEEE Vol. 67, No. 12, pp. 1624-2642.

It was very difficult for Hyperdex engineers to perfect its text encoding method. After many man-years of experimentation, Hyperdex engineers discovered a favorable technique to extract text data from text fields, an optimal assignment of binary codes having the advantage that key values occur with substantially different frequencies, and a way to organize the bits of the superimposed code words in storage such that only a small fraction of these bits need be retrieved and processed on each query. This substantially reduces the time required to execute a simple text query, and since the text bitmaps are stored congruent to all other bitmaps, eases their use in processing complex predicates.

Putting It All Together

When all the elements are put together we get the image shown below. The complete collection of bitmaps, of all indexed data types, are combined in the center element titled Hyperdex Index. The type of bitmap encoding used is dependent on the data type. The Bitmap Encoding Logic, shown on the left, is used by both the database indexing process and the query process.



A collection of bitmaps derived from the database and searched by using the same bitmap encoding logic that was used to create this structure to also transform query information to determine which bitmaps to search.

The Resultant Bitmap, shown at the bottom of the above figure, receives the result of a query. The Query Logic through the Bitmap Encoding Logic determines which bitmaps are retrieved from the index and the type of logical operations used to combine them. Only the selected bitmaps are retrieved speeding the query results.

The Resultant Bitmap is a bitmap like any other bitmap; i.e., a string of 1's and 0's. This bitmap has to be counted to find the 1's, and these 1's have to be converted to the RIDs (record IDs) used to access the corresponding table rows. ***Both Oracle and IBM provide a direct row access capability based on a provided RID. Microsoft does not!*** Consequently, when using Microsoft's SQL Server database, some way must be found to directly access a row when you know its relative position. To solve this dilemma, when it hyperdexes a table, *Hyperdex creates a key table (keydex) which contains all the table’s key values indexed automatically on an identity column*, **KeyID**. When a search is performed, the resultant bitmap is scanned and the RIDs are used to access the keydex and build a table of selected keys. A join to this table is added to the user’s query.

Thus is shown why Hyperdex is a marvelous improvement in database indexing. You get the decided improvement of bitmap indexing for any and all data fields, and you get the efficiency of bitmap retrieval speeds. An additional advantage when using Hyperdex’s numeric range encoding and low cardinality encoding is that count queries can be answered directly from the Hyperdex index without opening a database connection. This result is particularly useful when experimenting to define a useful query.

Overview Hyperdex User Interface

The ideal user interface would be a low-level subroutine interface available to software developers who's job it is to develop higher level end-user programs. In a world conditioned to networked computers this brings up the issue of remote procedure calls (RPCs); i.e. how to access a subroutine or procedure in another address space (commonly on another computer on a shared network). This kind of facility is necessary for shared program use and is the basis of distributed, client-server applications.

Some of the earliest work with RPCs was done on the Unix platform. *Many years ago, Microsoft began modularizing Windows and their Windows applications by breaking them into functional components with well-defined, "version safe" interfaces. The idea was to allow pieces of Windows and other applications to inter-operate. The name first given to this effort was "OLE", which stood for Object Linking and Embedding. OLE suffered nearly terminal birthing pains and developed a reputation for just being a bad idea. Undaunted, Microsoft renamed it COM for "Component Object Model". This was still the same old OLE, but Microsoft appeared to hope no one would notice. COM fared somewhat better, but it wasn't until Microsoft gave it the sexy name "ActiveX", and built it into virtually everything, that developers finally gave up trying not to use it.*

Somewhere along the bumpy road from OLE through COM to ActiveX, Microsoft's industry competitors began working on a distributed object system for Unix called Corba. Microsoft's object system was not distributed, but that mattered little. Microsoft quickly stuck a "D" (for Distributed) in front of COM to create DCOM, their Distributed Component Object Model. Then they crammed it into every version of Windows starting with Windows 98. Recently, Microsoft introduced two new cross-platform communications protocols, .NET Remoting and Web Services. Web Services operates under the umbrella of .NET Remoting.

As it turns out, when applications exist on computers of similar type and on the same network, DCOM works OK and the performance is adequate. However, DCOM has its drawbacks in the Internet connected world and/or when the system has to communicate through a firewall. .NET Remoting eliminates the difficulties of DCOM by supporting different transport protocol formats and communication protocols. This allows .NET Remoting to be adaptable to the network environment in which it is being used. Programmers have tried very hard to implement client-server applications using OLE, ActiveX, COM, DCOM and now .NET Remoting without much success. Why is it so hard to implement a simple, remote procedure and be able to call it?

Well, under the covers, the difficult part seems to be in marshalling the subroutine arguments between differing computer systems. In computer programming, **marshalling** *is the process of gathering data from one application's storage area, putting the data pieces into a message buffer, and organizing or converting the data into a format that is prescribed for a particular receiver or programming interface*. This is difficult for many programmers to understand and even harder for them to carry out.

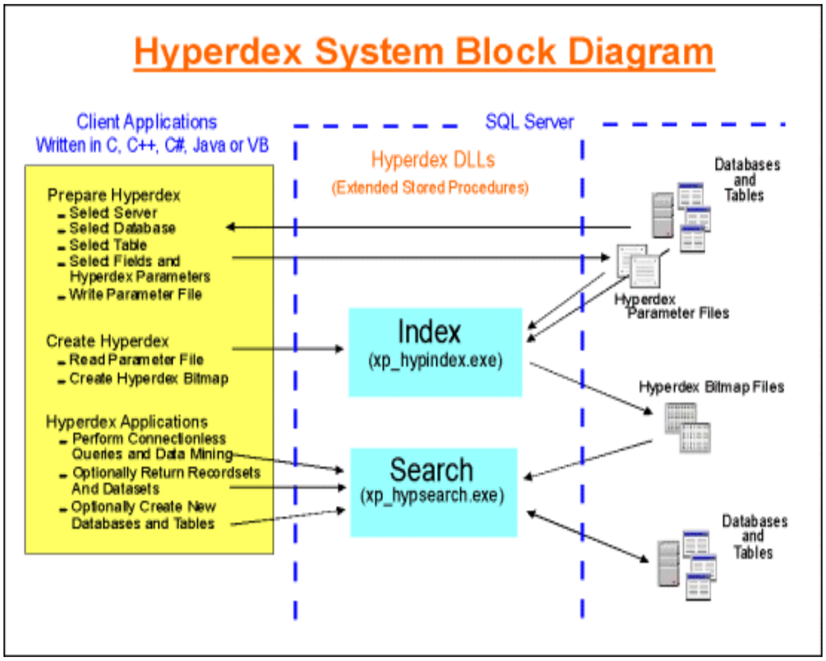
However, most database vendors provide mechanisms to build and store software procedures internally on their database systems and to enable programmers to call and use them. These server-based software routines are called Stored Procedures. Stored procedures are easy to use, have a well-defined call interface, and have long played a significant role in client-server applications. However, they are programmed in a pseudocode that is interpreted by the database system, and for this reason some find them slow and inefficient.

Microsoft's SQL Server supports a form of stored procedure called an Extended Stored Procedure (ESP). An ESP can be written in C, C++ or Delphi, which gives it the performance profile of a compiled code module. In all other ways it looks like and performs like a regular database stored procedure.

The developers of Hyperdex were looking for a simpler software interface to that provided by OLE, ActiveX, COM, DCOM and .NET Remoting. They recognized that Microsoft has a major commitment to SQL Server and its role in client-server development. Thus, Microsoft SQL Server engineers had to simplify how users connect and utilize SQL Server including how stored procedures are called. Hyperdex engineers decided to ride along on SQL Server's proven safe and reliable database interface by development of Extended Stored Procedures to wrap its Hyperdex subroutines.

Hyperdex System

The **Hyperdex system is designed to be database independent**, but is demonstrated below in the context of Microsoft SQL Server.



The Hyperdex components are the two blue boxes in the center of the figure. They are implemented as SQL Server Extended Stored Procedures as discussed below. The yellow box on the left shows the role of a client application. The area delineated by the blue dotted lines shows SQL Server and that the Hyperdex components are incorporated into the database. The right side of the figure shows the relationship of the databases and tables and the Hyperdex technology.

Hyperdex Extended Storage Procedures

Having established that the Extended Stored Procedure (ESP) is the most ideal way to present Hyperdex's user interface, the following will describe that interface. There are two interfaces of interest: (1) the Index ESP, and (2) the Search ESP. The two ESPs are accessible through any of the common programming languages, Visual Basic, Java, C#, C and C++. Since the Hyperdex ESPs are themselves written in C++, they can be easily and quickly implemented for any database manufacturer's environment.

Index ESP

The Index extended stored procedure (xp-hypindex) is used to create a Hyperdex index for a selected table. It is called passing two parameters, (1) the database name, and (2) the name of the table to be indexed. Prior to calling xp_hypindex the calling program must first create a parameter file within the database being used. The parameter file must have the name of the indexed table appended with the suffix _*hyp*. For example, if the table was named Customers, the parameter table would be named Customers_hyp.

To avoid having to pass a variable number of parameters to the xp_hypindex subroutine, a parameter file is used. Identified within the parameter file is the database name, the table name, and a list of the data fields to be hyperdexed. The data field names should have the same names as the table's field names. Included should be the SQL description of each field, the SQL type, and the field size. Also required is a field ID (an integer value synonymous with the field's offset in the record) and the Hyperdex type of the field. The field ID is used to uniquely identify each field. The Hyperdex type is an assigned value to specify the kind of bitmap encoding to be used (NUMERIC, BIT, CHAR, VARCHAR, DISTINCT (low cardinality), or KEY (the table's primary key).

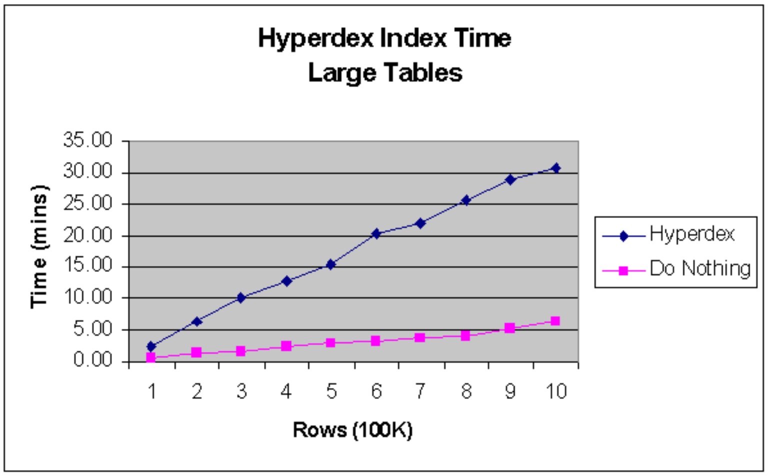
The parameter file also has an entry to identify where the new Hyperdex index is to be stored. This can be anywhere on the user's hard drive, but is typically stored in the Microsoft SQL Server folder in a sub-folder named Hyperdex. When the xp_hypindex routine is finished it places a file of Hyperdex Statistics(*tablename*.txt) in the Hyperdex folder. This file includes such things as the database name, the table name, the number of rows, the table size (in bytes), the index size (in bytes), the total number of fields in the table, the number of fields hyperdexed, the Hyperdex time to index (in seconds), the Hyperdex size (in bytes), and the Date/Time when the Hyperdex was created.

Included in the Hyperdex time is the time it takes to create a Hyperdex Keydex table. This table is indexed on an identity column (a column of integers from 0 to *n*, where *n* = the table size) and contains a list of the keys from the indexed table. This table is used when queries are performed on the indexed table and the resultant bitmap is used to determine the ones (hits). The position count of these ones is used to access the corresponding key in the Keydex table to determine the table row to retrieve. NOTE: This procedure and the Keydex table are not required with database vendors (Oracle, IBM and others) that have access capability using row IDs.

A demonstration program for the Hyperdex technology has been written. This program, written in Visual Basic .NET, shows how the parameters for a selected table are accumulated and how the xp_hypindex subroutine is called. This program has a user interface that permits a user to select the major function (Index, Search or Output) to be used. When Index is selected the user interface allows the server to be selected, then the desired database, then the table to be indexed, and last a selection mechanism to pick the data fields to be included in the hyperdex. When the xp_hypindex routine finishes it accesses the Hyperdex Statistics file and publishes it on the screen.

Hyperdex Index Performance

Some would wonder how much time does it take for Hyperdex to index a table. On small tables (10k to 100k rows) Hyperdex indexes about 1000 rows per second. On larger tables Hyperdex indexes about 30K rows per minute. This can be shown more graphically in the figure below.



NOTE: The results shown above were obtained on an Intel Celeron CPU 2.00 GHz processor with 500 MB RAM. The disk is a Western Digital WD400BB 7200RPM, 40 GB drive with 4.2 ms seek time.

Performance times are not relevant unless compared to some relative standard. In the figure above the "Do Nothing" curve is a plot of the time it takes the SQL Server Query Analyzer to do nothing, but simply load the table. Hyperdex has to load the table as well as index all the selected fields in all the rows. In the experiments that were run Hyperdex indexed 14 of 16 fields.

It is interesting to *note how Hyperdex scales when used with different table sizes*. As both figures above show Hyperdex is linear from 100K to 1000K rows both in speed of indexing and index size. The Hyperdex index is stored horizontally partitioned and processed in discrete sections, and there is no unlinear indication of scaling difficulty in processing time or storage. This is especially important for the large to very large fact tables used in Data Warehousing.

Search ESP

The Search extended stored procedure (xp-hypsearch) is used to query a selected table using its Hyperdex index. It is called passing a variable number of parameters, not all of which are required. The concept behind the Search ESP is to provide a simple interface to produce several types of results. The first subroutine argument is required and is a regular SQL statement specifying the query. It is a SQL statement like any other:

```
SELECT Cust_No, Customer_Name, City, State FROM Customers WHERE City = 'Jacksonville'
```

The second parameter is also required and is an integer that specifies the kind of output desired from the xp_hypsearch subroutine. The possible output values are:

- 0 - This type of output uses only the SQL statement and returns the number of hits in the bitmap. This type of query accesses only the Hyperdex index and does not open a database connection.
- 1 - This type of output uses the SQL statement and returns a recordset containing the selected rows retrieved from the specified table. The bitmap produced is stored in the tempdb database where it will be automatically deleted.

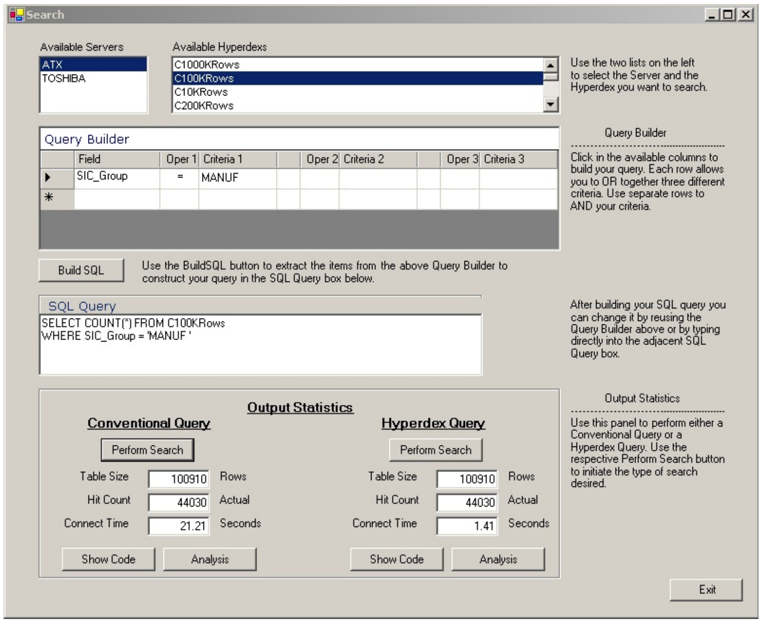
- 2 - This type of output uses the SQL statement and a specified Hyperdex subfolder in the SQL Server folder which it uses to store the bitmap file. An input parameter specifies the bitmap file name.
- 3 - this type of output uses the SQL statement and creates a new table in the specified database that contains the selected rows. Two input parameters specify the database name and new table name.

The action of the xp-hypsearch routine is to parse the SQL statement and identify the table(s) being accessed. If one of the tables has an associated Hyperdex index it is queried using the Hyperdex index to produce a bitmap and a keyset table. The output options listed above provide for various ways to use the Hyperdex results.

Using applications can select any of the Hyperdex output options to process a query. Since Microsoft does not permit using a Row ID directly, any using application specifically using bitmap output must access a keyset table by its identity column key to get the key(s) to the returned records.

Hyperdex Search Performance

The Hyperdex demo program was used to run a series of search speed trials. The demo was programmed using Visual Basic .NET and performed in such a way that conventional SQL Server access was used along with Hyperdex access. The figure below shows the relative performance.



Use the two lists on the left to select the Server and the Hyperdex you want to search.

Query Builder

	Field	Oper 1	Criteria 1		Oper 2	Criteria 2		Oper 3	Criteria 3
▶	SIC_Group	=	MANUF						
*									

Query Builder

Click in the available columns to build your query. Each row allows you to OR together three different criteria. Use separate rows to AND your criteria.

Use the BuildSQL button to extract the items from the above Query Builder to construct your query in the SQL Query box below.

SQL Query

```
SELECT COUNT(*) FROM C100KRows
WHERE SIC_Group = 'MANUF '
```

After building your SQL query you can change it by reusing the Query Builder above or by typing directly into the adjacent SQL Query box.

Output Statistics

Conventional Query

Table Size 100910 Rows
Hit Count 44030 Actual
Connect Time 21.21 Seconds

Hyperdex Query

Table Size 100910 Rows
Hit Count 44030 Actual
Connect Time 1.41 Seconds

Output Statistics

Use this panel to perform either a Conventional Query or a Hyperdex Query. Use the respective Perform Search button to initiate the type of search desired.

The query used for both test series was:

SELECT COUNT(*) FROM <table> WHERE SIC_Group = 'MANUF'

where the column SIC_Group in the test tables is a non-conventional key field of the CHAR data type. The <table> value was the names of 10 tables ranging in size from 100K rows to 1000000 rows.

Hyperdex search times averaged one half a second for all the tables searched. Microsoft's conventional database logic averaged 70 seconds. A few seconds is reasonable but queries taking 10, 20 seconds or longer seem to take forever when you are sitting at a computer workstation. On a query on a 1 million row table the Hyperdex time was 0.63 seconds. It certainly doesn't look like Hyperdex search speeds are increasing as a function of table size. This makes Hyperdex ideal for all kinds of read-only applications; IT, internet applications, business intelligence, data mining, and especially Data Warehousing with its large fact tables.

What Does Cardinality Mean?

In mathematics, the term cardinality refers to the number of cardinal (basic) members in a set. For example, the set A = {1, 2, 3} contains 3 elements, and therefore has a cardinality of 3. In SQL (Structured Query Language), the term cardinality refers to the uniqueness of data values contained in a particular column (attribute) of a database table, and is important in determining a query access plan.

When dealing with columnar value sets, there are 3 types of cardinality: *high-cardinality, normal-cardinality, and low-cardinality*.

High-cardinality refers to columns with values that are very uncommon or unique. High-cardinality column values are typically identification numbers, email addresses, or user names. An example of a data table column with high-cardinality would be a USERS table with a column named USER_ID. This column would contain unique values of 1-*n*. Since the values held in the USER_ID column are unique, this column's cardinality type would be referred to as high-cardinality, C >50.

Medium-cardinality refers to columns with values that are somewhat uncommon. Medium-cardinality column values are typically names, street addresses, or vehicle types. An example of a data table column with medium-cardinality would be a CUSTOMER table with a column named LAST_NAME, containing the last names of customers. While some people have common last names, such as Smith, others have uncommon last names. Therefore, an examination of all of the values held in the LAST_NAME column would show "clumps" of names in some places (e.g.: a lot of Smith's) surrounded on both sides by a long series of unique values. Since there is a variety of possible values held in this column, its cardinality type would be referred to as medium-cardinality, 16<C<50.

Low-cardinality refers to columns with few unique values. Low-cardinality column values are typically status flags, boolean values, or major classifications such as gender. An example of a data table column with low-cardinality would be a CUSTOMER table with a column named NEW_CUSTOMER. This column would contain only 2 distinct values: Y or N, denoting whether the customer was new or not. Since there are only 2 possible values held in this column, its cardinality type would be referred to as low-cardinality, C <= 16.

Most database manufacturers recommend that bitmap indexing be restricted to low or medium cardinal data fields.

Why are Fact Tables So Large?

Fact tables are the central tables in a data warehousing schema. Fact tables typically contain facts, usually numeric data, that can be summarized to provide information about the history of the operation of an organization. Fact tables contain little to no descriptive data; this is stored in the dimension tables that link to the fact table.

Fact tables contain only facts and foreign keys to the dimension tables. Dimension tables are usually textual and descriptive and contain the information used in queries and as the row headers of result sets. Usually, the composite of the foreign keys from the dimension tables are defined as the fact table's primary key. Queries typically join dimension tables to a fact table. Without some kind of intelligent assist all rows in a fact table are processed before a join operation with the dimension tables removes the non-qualifying rows. This is where bitmap technologies like IBM's Encoded Vector Index (EVI), Oracle's Bitmap Join, or the Hyperdex technology can come into play.

Unlike tables that are used in OLTP (On-Line Transaction Processing), fact tables typically store operational data accumulated over long periods of time. Because this history can result in many rows of data, care must be exercised in designing dimension and fact tables. Fact tables can become very large encompassing as many as millions of rows of data.

What is a Bitmap Join?

A Bitmap Join is an innovation owned and copyrighted by Oracle. As its name suggest it's a bitmap index, but one that is described through a join query. It's an index where the indexed values (attributes) come from one table but the bitmaps point to another table. With regard to data warehousing the index values might come from a dimension table and the bitmaps would point to the fact table.

Hyperdex Roadmap

Hyperdex has major potential specially with the emergence of the Cloud and Big Data. After reviewing the source code it would need some work to work in a Linux machine and implement it to our [redacted] project. This technology has great potential specially in the area of artificial intelligence down the road linked [redacted] suggested by [redacted].

[redacted]

[redacted] or possibly developing a custom database written in [redacted] to enhance the distributed properties of the technology. [redacted]

Note: The research donated and the lectures given by Joe and the things [redacted].

Fastcomcorp Research Team

Joseph Richburg
Francisco Pinochet
Eric Gough
Kimlong Loung
Michael Thompson
Ryan Sema