A Method for Searching Encrypted Ranges Using Comparator Values Generated by Binary Trees

Justin Parr

v1.2, December, 2022

JustinParrTech.com

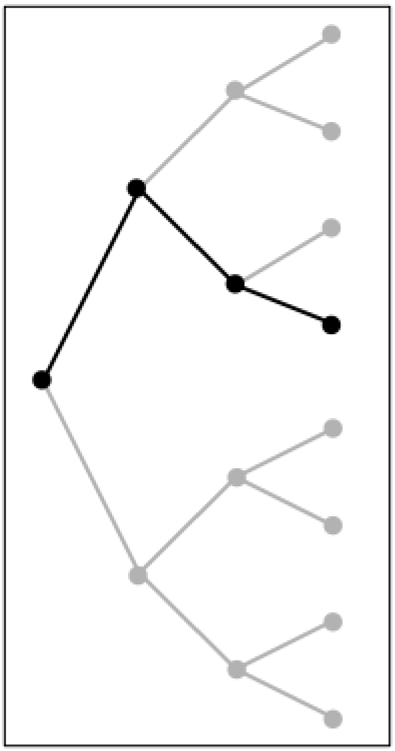

Table of Contents

1 Revision History

Version	Updates	Author
1.0 Nov, 2022	• Initial Version	Justin Parr
1.2 Dec, 2022	• Clarify B-Tree orientation in diagrams • Added: Comparison of “Binary Comparators” to other schemes in greater detail • Streamlined some of the wording • Added: Using other data types for comparators • Added: Deleting records (managing the B-Tree) • Modified watermark (page background)	Justin Parr

2 Abstract

A ‘comparator’ is a numerical value that is congruent to a data element’s ordinal value, but generated without disclosing any information.

One or two relational search terms can be passed to a database, and by converting the search terms to comparator values, range comparisons can be made on encrypted values based on comparator relationships to the underlying data.

Comparators are generated using a binary tree, which preserves the ordinal relationship of each data element, and can then be quantified as an integer value.

3 Overview / Executive Summary

Although it's widely agreed that field-level data encryption provides the best protection against data breaches, it also limits an application's ability to perform ranged searches, where an inequality comparison is performed against search terms.

Although there are existing strategies which use search trees, these are more focused on managing decryption keys than the search itself.

Assigning artificial search keys is not a new approach, but the general problem is that it can result in information disclosure as well as key collisions. If the keys are regularly-spaced, this can lead to an inference about the underlying data values, and if the spacing is too narrow, this can lead to insufficient search keys when presented with a large quantity of data values for given key interval.

In the scheme proposed herein, a binary tree is used to generate integer search keys, called comparators, that are non-sequential but maintain the same ordinal relationship as the underlying data. Because comparators have no fixed relationship to each other, they don't leak any information.

4 Encryption and Databases

Encryption scrambles data in a predictable way, such that it can be decrypted (or unscrambled) later. This is an important tool that helps protect against a data breech, where an attacker is able to access a database where sensitive data is stored in bulk.

In an encrypted state, sensitive information such as names, addresses, birth dates, credit card numbers, bank account numbers, and contact details are worthless when stolen by an attacker.

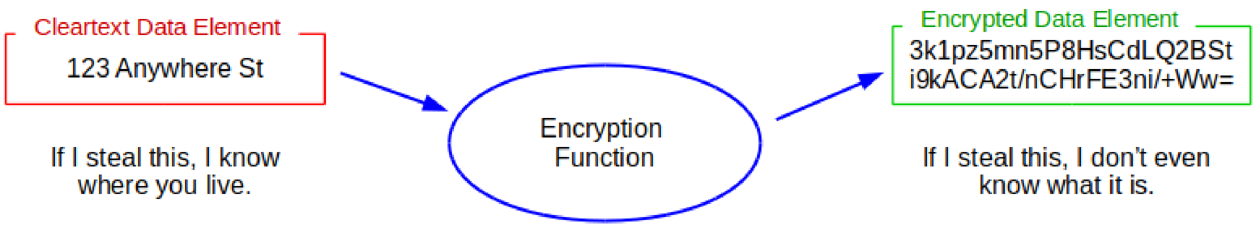
Cleartext Data Element
123 Anywhere St
If I steal this, I know where you live.
Encryption Function
Encrypted Data Element
3k1pz5mn5P8HsCdLQ2BSt i9kACA2t/nCHrFE3ni/+Ww=
If I steal this, I don't even know what it is.

Illustration 1

Using OpenSSL to encrypt an address using the password "password" and no salt.

```
echo 123 Anywhere St|openssl aes256 -e -k password -nosalt | base64
```

```
To decrypt:
```

```
echo 3k1pz5mn5P8HsCdLQ2BSti9kACA2t/nCHrFE3ni/+Ww=| base64 -d|openssl aes256
-d -k password -nosalt
```

When not encrypted (in a cleartext state), an attacker can use these details to commit fraud and identity theft, and data breaches account for billions of dollars lost to fraud every year.

Encryption can be implemented in a number of different ways, but experts generally agree that field-level encryption is the most secure.

Type of Encryption	Technology Layer	Implication
Field-Level	Application	Encryption is managed by the application, and data is only presented in cleartext when the application is using it.
Column-Level	Database	The Database Management System (DBMS) manages the encryption and decryption. Data values are stored in an encrypted format, similar to field-level encryption, but data values are passed to the application in cleartext (although typically using an encrypted transmission channel)

Transparent Data Encryption (TDE)	Database	The DBMS stores the field values in cleartext, but the database file itself is encrypted by the DBMS as it’s written to disk. As with column-level encryption, TDE is transparent to the application.
Encrypting File System (EFS) / Encrypted File System / Block-Level	Operating System	The Operating System is configured to encrypt specific files, folders, or an entire partition. The encryption is managed by the OS, and is transparent to both the database and application.
Storage-Level / Block-Level	Virtualization / Storage	The Operating System runs in an environment where the hardware and / or storage are virtualized. Encryption is managed by the Storage Controller and / or Hypervisor, and is transparent to the OS, database, and application.

As the encryption moves farther from the application, it becomes faster and less expensive because some other layer is handling the encryption and decryption.

However, each layer that presents the data in cleartext also presents an attacker with an opportunity to steal it. For example, if using block-level encryption and the OS is compromised, an attacker has access to the entire database file in cleartext, which could be exfiltrated in-tact, and dissected later.

Field-level encryption, while it has the most overhead, is also the most secure because only the application sees the data in cleartext.

Within the application, searching for a specific, encrypted value (equality search) is relatively straightforward:

- Because the encryption process is deterministic, a given cleartext data value will always result in the same encrypted version of that value.
- Assuming that the user enters a specific search value, the application encrypts it, and then performs a database search using the encrypted value.
- Again, because the encryption process is deterministic, if the encrypted value exists in the database, then the cleartext search value is the same as the cleartext version of the encrypted value found in the database.

However, a byproduct of the encryption process is that because numbers, dates, and strings are scrambled, they lose their ordinal relationship. This makes ranged (inequality) searches impractical unless the application decrypts and compares every value against the search value.

For example:

- Searching ranges of encrypted birth dates, account numbers, or account balances
- Searching ranges of encrypted health data, such as blood pressure or heart rate
- Searching ranges of encrypted text data, such as names beginning with "J"

What would normally be a simple ranged search, such as "WHERE birthDate > 1/1/1990" is no longer feasible because every birthDate in the database is scrambled, and has no ordinal relationship to the search value. Therefore, each birthDate must be retrieved by the application, decrypted, and then compared to the search value, '1/1/1990'. The same is true of ranged text searches, for the same reason.

For clarity, we will differentiate the two following, unrelated terms:

- **Encryption Key**: Allows an encryption algorithm called a cipher to encrypt or decrypt data. The encryption process takes the data and key as input, and scrambles the data in a deterministic way, so that it can be unscrambled later using the appropriate decryption key. If the wrong key is specified during the decryption process, the data remains scrambled.

- **Search Key** / **Database Key**: Allows a database engine to identify a specific row or rows based on a unique value. For example, a "people" table might have a unique key called "PersonID". Even if there are multiple people with the same name, each would each have a unique PersonID. To tell the database to retrieve, update, or delete a specific person, you would specify the PersonID rather than the name.

Although there are existing strategies which use search trees, these are more focused on managing the decryption keys than the search process itself.

Assigning artificial search keys is not a new approach, but the general problem is that it can result in information disclosure as well as key collisions. As an example, given sequential search keys, and knowing some of the underlying associated values allows an attacker to deduce other values by induction.

For example:

> If we have encrypted street addresses, where k is the set of search keys, k_{10}=1230 and k_{20}=1232, then it's reasonable to assume k_{15}=1231.

The other problem with this approach is that, despite the interval gap, there is no way to predict the relative ordinal value of new data added to the database, whose quantity might exceed the gap.

For example:

> If we have encrypted street addresses, where k_{10}=1230 and k_{20}=1250, what happens if we add all of the street addresses between? There are 19 possible data values in the range, but only 10 search keys in the key interval, which leads to the possibility that this scheme could have to deal with 9 data values without search keys, or would have to re-allocate and shuffle search keys, etc.

4.1 Impact to Database Indexes

Each data table within a database may have one or more indexes, and the purpose of an index is to maintain a sorted, searchable list of pointers to the data. This can be used to speed up searches and sorts for frequently-used columns, especially when searching key fields that are used for joining tables.

Of course, the Database Management System (DBMS) doesn't *physically* rearrange the data – an index works like a linked list that maintains the proper sequence when the DBMS is asked to return data from that particular table.

In addition to facilitating equality searches (a search for a specific value), indexes speed up ranged searches because each index maintains a specified order for the affected data elements.

For example:

> Creating an index on a column called 'date' would allow the DBMS to quickly return a list of dates within a specific range without having to scan each row of data (called a table scan) which is both expensive and slow. Instead, the DBMS consults the index, which already maintains a sort order for 'date', and can simply do a binary search within the index to find the top and bottom of the range, and then return all the rows in between.

Likewise, an application can request the most recent transaction by asking the database for the largest number from an integer column called 'transaction_ID', and the DBMS finds this almost instantly by simply jumping to the end of the appropriate index.

However, if the indexed column is encrypted, this largely negates the value of the index because the values are no longer sorted properly.

5 Comparators

In the proposed scheme, every unique encrypted data value has a corresponding, unique, integer value called a **comparator**. The purpose of the comparator is to maintain the ordinal relationship of the underlying data. When new data is added, a comparator is generated for each new data element.

For searching, a user passes one or more search parameters to the application. The application generates a comparator for each search term, and then performs a regular database query using the comparator values. Because the comparators maintain an ordinal relationship to the data, this allows the database to perform both equality and ranged searches.

As a result of the query, the database returns a data set containing the relevant comparator values for each record, along with the corresponding encrypted data elements, which are then decrypted as needed by the application.

5.1 Generating Comparators

Comparators are created using a binary tree whose nodes are the underlying cleartext data values. At each step, the application retrieves an encrypted data element (node) from the database, decrypts it, and performs a comparison.

- The application selects a data element to be the "pivot", which is the first element in the tree. The pivot is assigned the address of "1".
- As new elements are added, the binary tree is built in the usual fashion.
    - If the new value is greater than the decrypted node, the application descends to the right-hand child node.
    - If the new value is less than the decrypted node, the application descends to the left-hand child node.
    - If the new value matches a node, it shares the same address as the matching node.
    - If the application finishes descending in to the tree without finding a matching node, the new value gets added as a child of the last node it visited – either as the right-hand or left-hand child based on whether it is greater or less than the decrypted node.
- The tree is a logical structure, and the nodes pointed to by the tree are the encrypted data elements, which are only decrypted when traversing the tree.
- The address of each node is the path taken through the tree, plus "1".

- Starting at the pivot, the address is empty.
- For each left-hand turn (value is less than the node), a "0" is added to the address.
- For each right-hand turn (value is greater than the node), a "1" is added to the address.
- A "1" is added to the end of the address.

- **The resulting comparator is the address, left-justified in an n-bit integer field.**

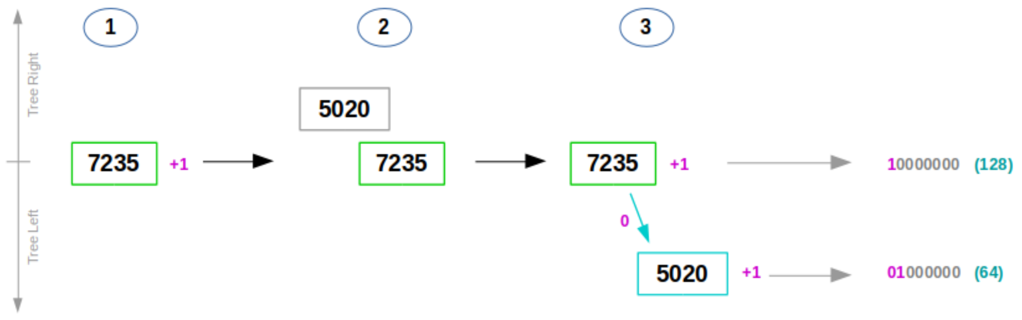

Illustration 2

The initial process of building an example binary tree. Data values are shown in cleartext, but would be encrypted in the database and only visible to the application.

For ease of depiction, all binary trees are shown rotated to the left – right-hand nodes progress upward, and left-hand nodes progress downward.

In Illustration 2, we start with a pivot value, 7235 (1). The next value, 5020 is evaluated against the pivot (2). In (3), 5020 is less than 7235, so it gets added as the "left" child node. Each node's address is the path through the tree, plus "1".

- Our pivot, 7235, has address "1", which is "" (blank) with "1" appended.
- Our next data value, 5020, is less than 7235, and gets added as the "left" child node. Tracing the path through the tree for 5020, we start at the pivot (current address=""), go left (current address="0") and then append "1" (final address="01")
- If we left-justify both of these addresses in an 8-bit integer field, we get 128 (1000000) and 64 (01000000) respectively

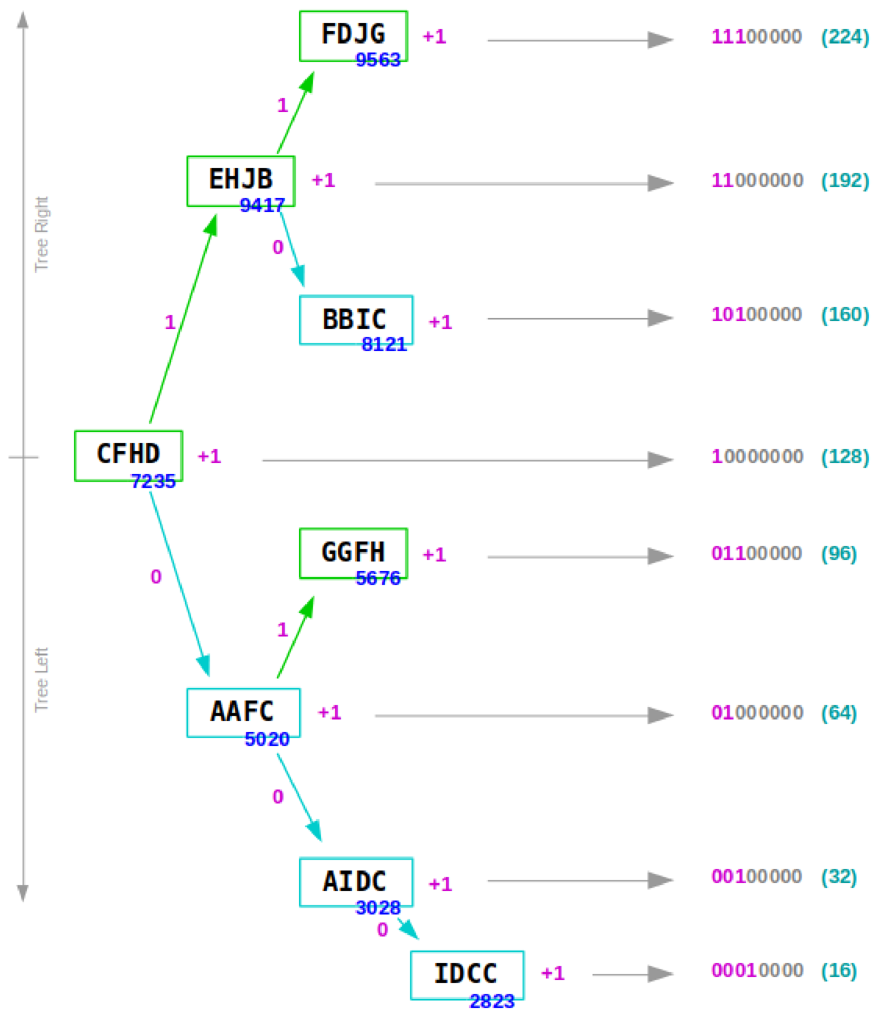

Illustration 3

After a few more nodes have been added, this is what the tree looks like.

For ease of depiction, the tree has been rotated to the left – right-hand nodes progress upward, while left-hand nodes progress downward.

In Illustration 3, the data values have been replaced with their encrypted counterparts. In reality, all database operations are conducted in an encrypted state.

Looking at the comparator values, the necessity of adding a “1” at the end of the address becomes clear. Without it, the left-hand three nodes would all have the same address:

Data Element	Tree Address	Without “1”	With “1”
5020	0	00000000 = 0	01000000 = 64
3028	00	00000000 = 0	00100000 = 32
2823	000	00000000 = 0	00010000 = 16

The trailing “1” ensures that each comparator is discreet.

5.1.1 Irregular Distribution Prevents Data Leakage

As we’ve seen, schemes that use a fixed search key spacing can leak data by allowing an attacker to analyze search keys, and then interpolate the underlying data values.

This is possible because the search keys maintain a linear relationship to each other, and to the underlying data.

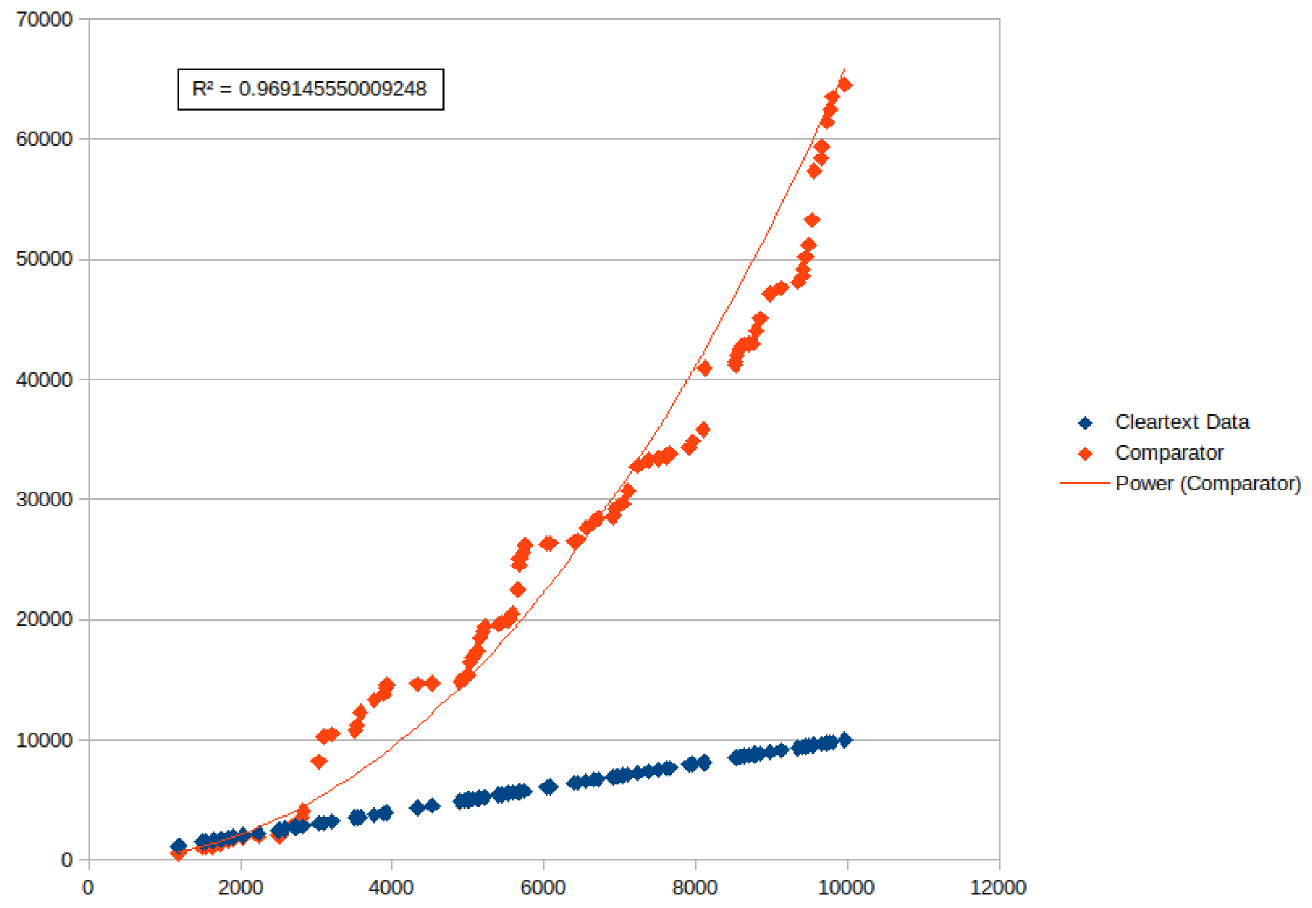

Illustration 4

For cleartext values N, the distribution of comparators progresses at approximately N^2, but the distribution is irregular, and therefore complicates any attempt to interpolate the underlying data.

Comparators, despite being ordered, occur based on their relationship within the tree rather than at specific intervals, making them much harder to predict.

Although the distribution of comparators progresses at approximately n^2, any new comparator can appear anywhere between two existing ones. Rather than being exactly between them, it could be arbitrarily close to either one. And, despite this, the scheme always allows new comparators to be inserted between two others, regardless of how close they are.

This works because the tree address is left-justified within a bit field, making it possible to use subsequent bits. Despite being an integer, the comparator behaves similar to a decimal, where you can always create a new number that's between two others by simply adding another decimal digit.

For example:

> Creating a number between 0.345 and 0.346 can be accomplished by adding a digit. All of the numbers between and including 0.3451 and 0.3459 are between the two. And for each pair, there are an infinite number of numbers between them.

The comparator scheme works in a similar fashion, for as many bits that exist within the integer bit field.

5.2 Tree Capacity and Scaling

If n is the number of data elements predicted to occur in the set, the size of the largest comparator is approximately n^2. Because the tree is binary, $\log_2(n^2)+1$ bits are required – the extra bit accommodates the “1” address terminator.

Bit Field Size	Approximate Number of Elements
8 (7)	11
16 (15)	181
32 (31)	46,340
64 (63)	3,037,000,500

Counter-intuitively, the size of the comparator only depends on the number of elements, not the magnitude of the underlying data value. Therefore, even large data elements such as long strings can be represented concisely by integers.

Although either a 32-bit or 64-bit comparator would be suitable for most applications, the comparator could consist of multiple integers or a string of integers. Chaining an arbitrary number of integers would allow for a virtually unlimited tree size.

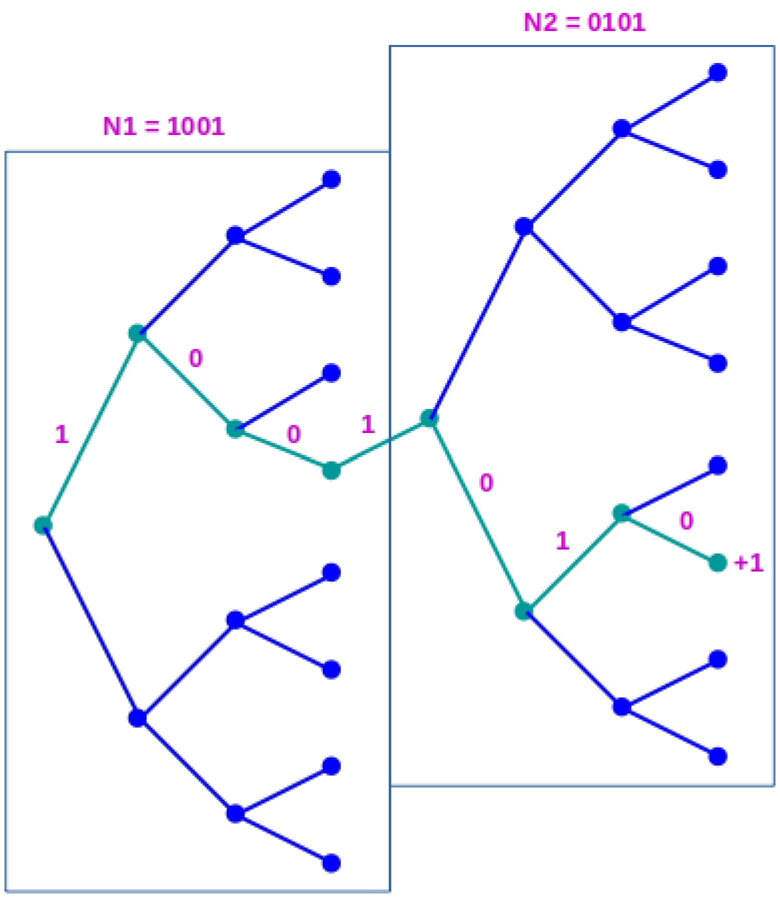

Illustration 5

Here, two nibbles (4-bit field) are chained together, which is effectively a single 8-bit field.

For example, a chained 128-bit (127) comparator would consist of 4 chained 32-bit integers or 2 chained 64-bit integers, and would be capable of representing about 13 quadrillion data elements.

5.3 Tree Balance, Refactoring, and Collisions

If a specific area of the tree becomes overpopulated, this results in excess depth, and could lead to a condition where new values can no longer be inserted.

5.3.1 Refactoring

One method used to address an unbalanced tree is refactoring.

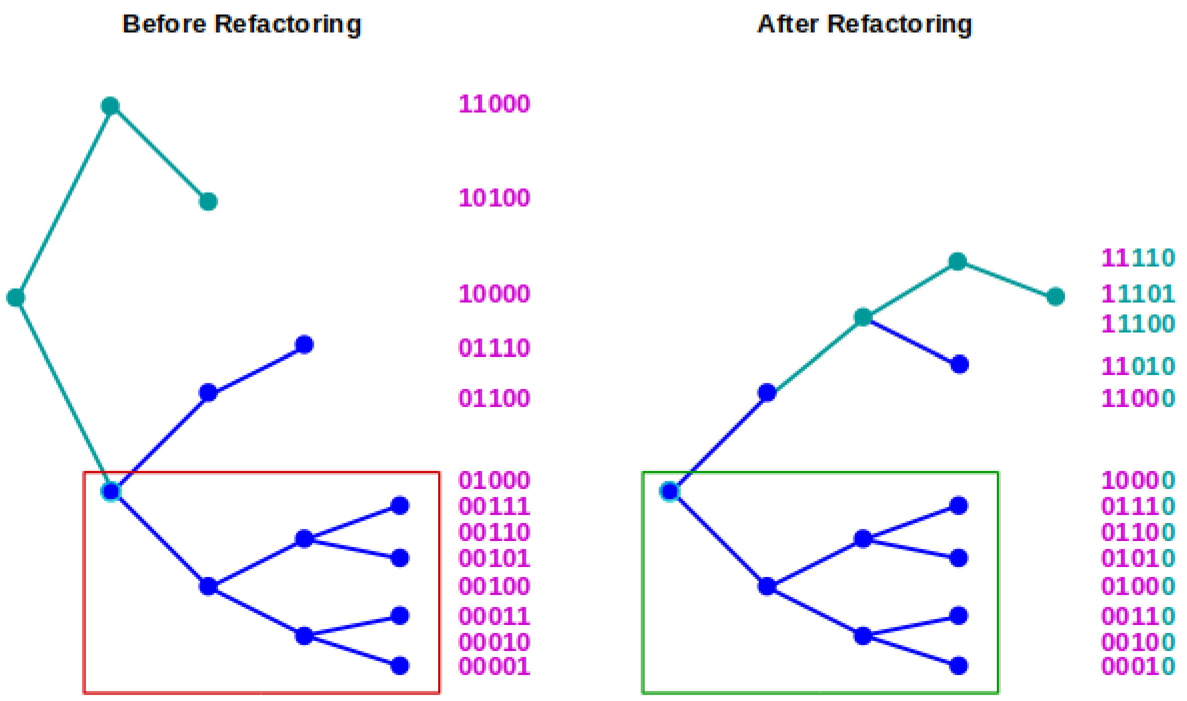

Illustration 6

On the left, the tree is unbalanced, resulting in crowding within the red area – in this example, there is no room to add new comparitors, because doing so would exceed the bit (tree) depth.

On the right, the tree has been refactored by selecting a new pivot. Because each node in the crowded section is one position closer to the new pivot, this effectively doubles the capacity within that portion of the tree. The previous pivot and its child nodes have been re-grafted to the new pivot.

As there are many algorithms for accomplishing this, and all are well documented elsewhere, a discussion of the specifics will be excluded here. In general, these algorithms select a new pivot (root), and then re-graft orphaned nodes within the new sub-tree.

However, the end result of refactoring is that the tree is more balanced, where each node has approximately the same number of descendants on both child branches. Because the descendant nodes are better distributed, this results in a more shallow overall depth, which frees up capacity in regions that were formerly crowded.

This approach is elegant, because refactoring can be performed using the old comparators without having to actually decrypt any of the nodes' data values. During the refactoring process, new tree addresses, and therefore new comparator values are generated for all data nodes, and the old ones are simply discarded at the end. Because decryption is not required, the refactoring process can be conducted by the database engine, for example, within a stored procedure.

This is similar to the process for re-indexing a database table, because database indexes also use a tree structure. In fact, the final step after refactoring would be to rebuild any database indexes using the new comparator values, so that the new database index is also balanced properly.

5.3.2 Collisions

Without refactoring, the other option is to accept **collisions**, where a single comparator could relate to multiple data values, even though the underlying data values are different.

In some cases, it may be acceptable to allow collisions, understanding that this affects how queries behave.

For example, given the following data and associated comparators, we can analyze how allowing collisions would affect each type of operation.

Comparator (Collisions)	Comparator (No Collisions)	Cleartext Data Values
10	1 2 3	1111 2222 3333
40	4 5	4444 5555
60	6	6666
70	7 8 9	7777 8888 9999

In the table below, "c" is used for the comparator, and "A" and "B" represent data values.

Operator	Without Collisions	With Collisions
Equality (A=B)	Only exact matches are returned.	Values within a small range of A

	For example, WHERE c=4 ONLY returns 4444.	may be returned. For example, WHERE c=40 returns 4444 and 5555, but WHERE c=6 only returns 6666. If the application must make provisions for situations where the database returns multiple records.
Greater Than (A>B) Less Than (A<B)	Works as expected. For example, WHERE c>1 returns 2222, 3333, 4444, and so on.	Could result in gaps. For example, WHERE c>10 skips 2222 and 3333, and returns values starting at 4444 To avoid this, the application must issue >= (Greater or Equal) or <= (Less or Equal) and then filter out equalities. Likewise, WHERE c>=30 would revert to c>=10, and would improperly include 1111 and 2222, despite the fact that the application doesn't expect these values.
BETWEEN	BETWEEN works like a pair of inequality comparisons: A BETWEEN B_1 AND B_2 Is the same as: (A >= B_1) AND (A<=B_2)	Just as with >= and <=, some extra data values may be improperly included.
IN	IN works like multiple equality comparisons: A IN (B_1, B_2, B_3) Is the same as: (A=B_1) OR (A=B_2) OR (A=B_3)	Just as with =, some extraneous data values may be improperly included.

Depending upon the nature of the application, collisions and their associated limitations may be acceptable, in order to avoid refactoring or integer chaining.

5.4 Searching Using Comparators

In the illustration below, the original data values are shown in blue for clarity, but in a real database, the cleartext data wouldn't be present.

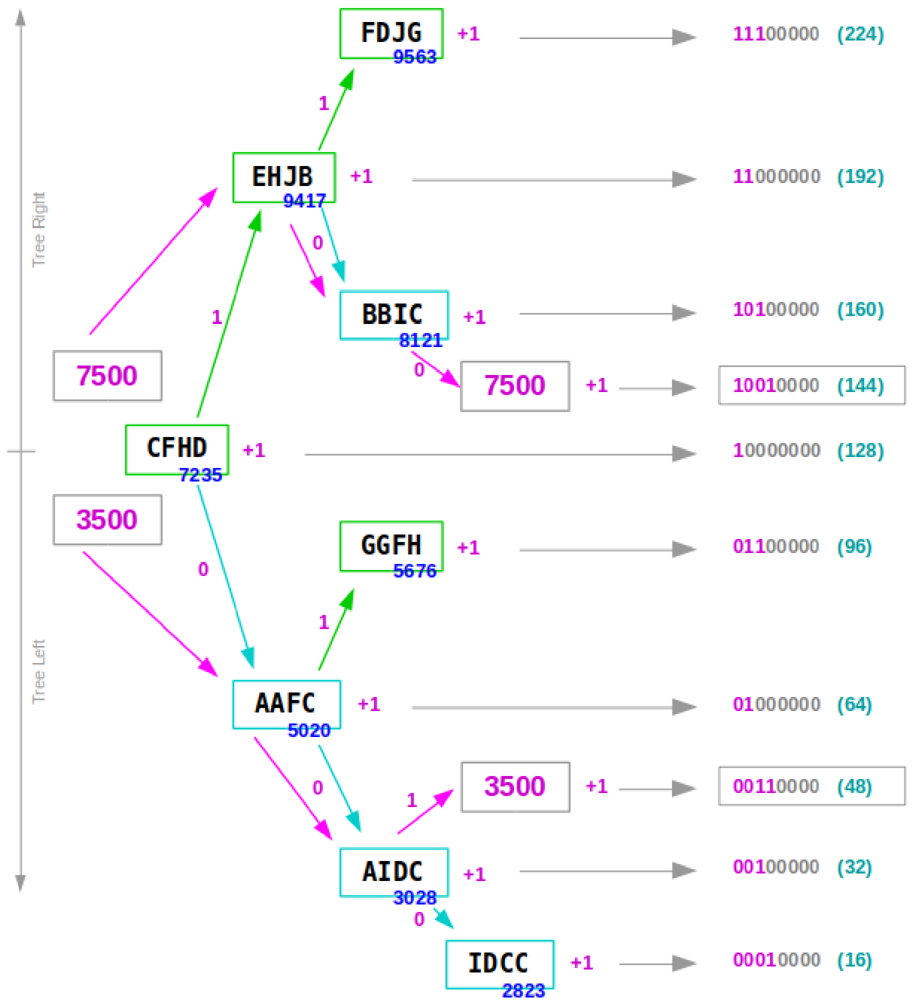

Illustration 7

Creating comparators from search terms follows the same process as adding a new node, but without permanence. When the search is complete, the search comparators are discarded.

In Illustration 7, we create comparators for two numeric parameters that are search terms for a BETWEEN operation in a WHERE clause.

The resulting search comparators, 48 and 144, maintain an ordinal relationship to the underlying data, even though we can't see the actual values.

Rather than having to decrypt every value in the database, we simply let the database perform a normal search on the comparators.

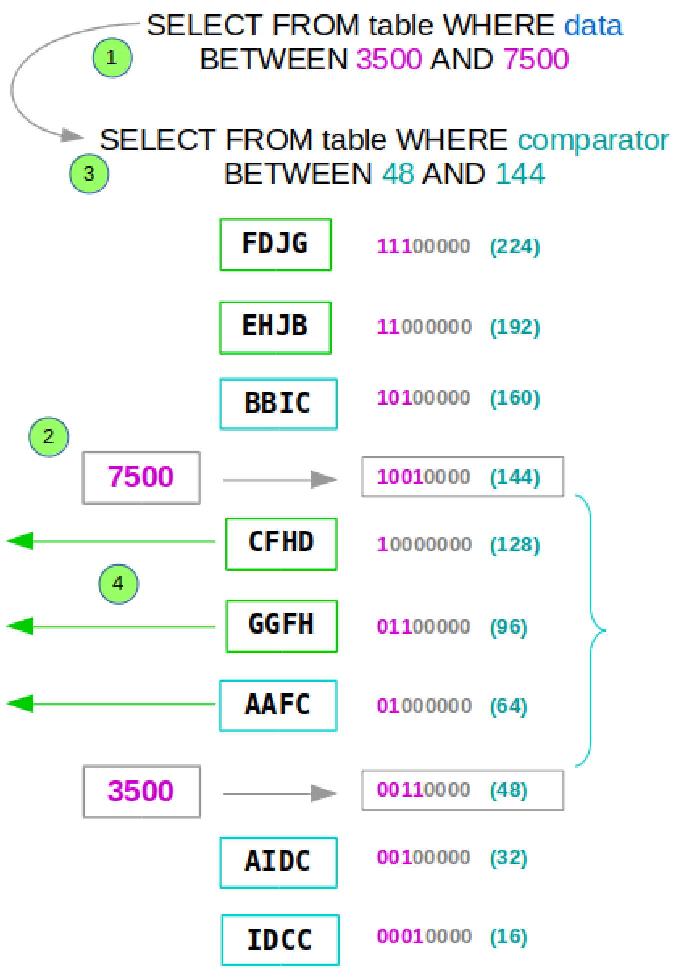

Illustration 8

The application replaces database search terms with search comparators, and then the database executes the query.

In Illustration 8, the application wants to run a query for data elements within a specific range. These could be confidential performance data for a new engine, or maybe protected health data.

1. The application needs to run a query using normal terms.
2. The application converts the search terms to search comparators.
3. The application writes a new query using search comparators.
4. The database executes the comparator query, and returns the resulting encrypted data elements to the application.

Once search values are converted to search comparators, the database query process behaves very similar to querying the underlying cleartext data.

However, the data returned to the application by the database could vary based on the use case.

Result Set	Use Cases
A list of comparators, assuming that comparators have a UNIQUE constraint (no duplicates are allowed)	If the application needs to identify certain records without actually acting on them, the application can act upon the comparator is if it was the underlying data, without having to decrypt it. For example, perhaps you want to send birthday cards to everyone born this month – the application can act on all records whose birth date is within a range without decrypting the actual birth dates. Another example is updating a group of accounts whose account number falls within a certain range, without ever decrypting the account numbers.
A data set containing encrypted data values	If the application needs to act upon the the data directly, the database can return the encrypted values, which are then decrypted as needed by the application. For example, a healthcare application might look for a patient's encrypted blood pressure readings that are within a certain range. Because the readings themselves have specific meaning, the application needs the actual data rather than just the comparator.
The result of an aggregation function, such as MIN, MAX, or COUNT	Aggregation functions can be used to find the oldest account, or simply tally the number of customers within a certain age range. In another example, an application can answer a simple question such as "is the account holder a minor" by using

	comparators. The application takes today's date, subtracts 18 years, and generates a comparator. If the comparator for a person's birth date is less than this value, they are a minor.

In addition to the above, and similar to data masking, comparators can be passed to downstream applications without compromising the underlying data.

5.4.1 Complex Queries

An application can query multiple, protected parameters, assuming that each one has an equivalent comparator.

For example, an application might need to know which account holders are within a certain age range, and whose credit scores are within a certain range. Both of these are protected elements, and therefore encrypted within the database.

However, if both birth date and credit score each have comparators, a complex query allows searching within an overlapping range:

SELECT FROM accounts WHERE c_dob BETWEEN $c_{1,1}$ AND $c_{1,2}$ AND c_credit_score BETWEEN $c_{2,1}$ AND $c_{2,2}$

In another example, if an account holder's address is encoded with latitude and longitude, each encrypted, and each having comparators, then a complex query can be used to search for customers within a specified geographical area:

SELECT FROM accounts WHERE c_latitude BETWEEN $c_{1,1}$ AND $c_{1,2}$ AND c_longitude BETWEEN $c_{2,1}$ AND $c_{2,2}$

And, as with data masking, if key values have unique comparators, joins can be performed using only the comparators:

SELECT a.* FROM tbl1 a INNER JOIN tbl2 b ON a.c_account = b.c_account

This would come in to play, for example, if joining an "accounts" table to a "transactions" table without having to decrypt individual account values.

5.5 Deleting Rows – Managing the B-Tree

When a row is deleted from a database table, it's encrypted value might leave a hole in the b-tree, which prevents any new comparators from being generated beyond the "missing" node.

There are several ways that the application can deal with this:

- **Leave the node in-tact**. Although the node no longer corresponds to a "real" data value, it can still be used to generate new comparators. This strategy requires that the application must store all node values in a separate table. "Deleted" nodes can be eliminated by refactoring. This approach requires the least computational overhead.
- A nodes has up to two children. Removing a node can be accomplished by **promoting one child, and then grafting the other**. If the two children are "L" and "R", respectively, then:

  All nodes in the left-hand sub-tree are less in value than the lowest value in the right-hand tree, or:

  LR..R < RL..L

  Effectively, this means that if we promote the right-hand node, the "orphaned" left-hand node can be re-grafted as a child of the right sub-tree's lowest node. Unfortunately, this results in a new sub-tree with the combined bit depth of both former sub-trees, and higher bit depths are less efficient.
- **Refactor the new sub-tree**. Computationally, this is the least efficient option, but results in the most efficient tree (post-deletion).
- **Generate an artificial node**. The new node must follow this relationship:

  LR..R < N < RL..L

  Based on this relationship, the easiest way to generate a new node is to average the two:

  N = (LR..R + RL..L) / 2

  The application would calculate the new node value and store it in a separate table, similar to a scheme used for keeping deleted nodes.

  This approach is functionally-equivalent to keeping deleted nodes, while being computationally less-efficient.
- **Graft both child nodes to other parts of the tree**. If we ascend to the deleted-node's parent, P_1, and to that node's parent P_2, then one of the following relationships most likely exists:

  a) P_1L..R < (R,L) < P_2L..L

  b) P_2L..R < (R,L) < P_1R

  As with promotion, this approach could lead to excessive bit depth.

5.6 Using Other Data Types as Comparators

Integers are very efficient, because they are the native data type of the CPU – both the database server and the application server can process and store them without extra steps required to convert them back and forth between other formats. Also, integers require the least amount of storage compared to other formats, making them efficient to store in large quantities.

Integers are so efficient that they are used natively by the database engine – things like record IDs and index pointers are stored and processed internal to the database as integers. Even dates and times are converted to integers, where they are stored and processed natively, and only need to be displayed as a date or time when a user needs to see the data.

However, modern servers have an abundance of CPU, memory, and disk storage, making it feasible (although not quite as efficient) to use other data types as comparators.

5.6.1 String Comparators

Database engines are good at comparing strings, and like numbers, strings are usually human-readable.

For example, base64 encoding uses a 6-bit human-readable symbol set to store groups of 3 bytes as groups of 4 symbols. This allows base64-encoded data to be easily transmitted or copied to other systems via user-space operations, such as copying in to an e-mail.

Using a similar scheme, comparators can be stored as a string of human-readable symbols, where each character stores up to 6 bits of information, and is substituted from a symbol table during the encoding process.

If we compare this to a 64-bit integer, which can store up to 63 bits of comparator data, an 8-byte (non-Unicode) string can store only 48. However, unlike native integer types, a string can be extended to any arbitrary length, making it much more flexible for large data sets.

Further, the database engine already knows how to sort and index strings, which mitigates most of the performance penalty incurred by using a more complex data type.

Human-readability is useful for debugging and troubleshooting, as well as interoperability with other systems. Likewise, if an encrypted data element needs to appears on a report, depending on the use case, its comparator can appear in its place.

5.6.1.1 Caveat about Strings as Comparators

Although comparators don't directly leak information, a clever attacker could use the length of a string comparator to deduce the population density of the corresponding region of the tree.

For example:

- Longer strings indicate a greater depth, which indicates a higher node density.
- Shorter strings don't necessarily indicate anything other than the fact that the node in question is closer to the root of the tree.

This type of information could be used as an attack vector, or in conjunction with other information about the underlying data.

5.6.2 Binary String

Although human-readability is nice to have, it's not usually a requirement.

In a binary string, each element represents a byte or some other integer data type, with the caveat that many elements won't be printable. A binary string can contain up to 8 bits of data per symbol (non-unicode), or up to 32 if stored as a unicode string.

This provides the advantages and flexibility of a string data type, while storing the tree address for each node as efficiently as possible. A byte array stored as a string behaves like an arbitrary-precision, big-endian integer.

If configured properly, most database engines can sort and index binary strings. This is usually referred to as "sort order", and a field containing binary strings would require a binary sort order.

5.6.3 Poor Data Types for Comparators

Some data types make poor comparators.

5.6.3.1 Floating Points

There is no reason to use a floating point over an integer, and floating points come with special complexities because of their format.

Type	Bits	Sign Bits	Exponent Bits	Mantissa Bits
Single-Precision	32	1	8	23
Double-Precision	64	1	11	52

If the application uses a floating point directly, the maximum precision available (size of the mantissa) is only about ¾ of the total size. If used as a bit field, this can produce unpredictable results, such as:

- +0 or -0, which are separate numbers. Integers use 2’s compliment, meaning that there is only one 0.
- “NaN” or “Not a Number”, which is a predefined, invalid value. There is +NaN and -NaN.
- Values can vary from very large (positive exponent) to very small (negative exponent)

5.6.3.2 Binary Large OBject (BLOB)

Most databases support a “BLOB” (Binary Large OBject) field type, meant to store images or other binary (not human-readable) data.

Regular data is stored in “pages” within the database, where each page contains multiple rows.

However, BLOB fields are stored as a pointer to another page location that contains the actual data. A pointer at the end of each BLOB page allows them to be linked, which means that BLOBs can grow arbitrarily large.

BLOBs have several disadvantages:

- Typically, because a BLOB is just a pointer, BLOB fields can’t be indexed directly – the application would rely on other fields to contain metadata (such as an image name) that can be indexed using the normal process.
- BLOBs are slower than other data types because of the indirect storage process.
- A minimum of one, dedicated BLOB page is allocated for each row, which means that using BLOBs is extremely expensive from a storage perspective.
- BLOBs can’t be interpreted natively by the database – only by the application.

6 Binary Comparators vs Other Schemes

There are currently three basic approaches to searching encrypted data.

6.1 IBE / MRQED

Identity-Based Encryption (IBE) uses the data itself to generate decryption keys. IBE schemes do not guarantee any kind of ordinal relationship to the encrypted data.

MRQED is a scheme that uses AHIBE (Anonymous, Hierarchical IBE) to grant an auditor limited access to network logs that are already sorted and presumably indexed prior to encryption. The focus of MRQED is to provide a a set of keys that can decrypt the information in question without compromising the rest. Like Binary Comparators, MRQED uses a b-tree sorting function, but the nodes consist of integers at fixed intervals within the tree.

This approach works well for data that's static, such as audit logs. However, this approach has two problems when dealing with tables that are taking live transactions:

- The search keys *could* be used to interpolate underlying data values by repeatedly querying different ranges.
- A tree that uses fixed-interval search keys can quickly become unbalanced, where no new nodes can be added to a crowded portion of the tree. Because binary comparators are derived from the underlying data element's tree location, the tree can easily be refactored if needed, without having to regenerate the entire tree.

6.2 Fixed Search Keys

Fixed search keys (with or without a b-tree) have the limitations described above:

- The keys themselves could be used to interpolate underlying data values, especially via repeated queries.
- The fixed interval provides very little flexibility when adding data to a highly-transactional table.

6.3 Hidden Vector

A hidden vector is a piece of metadata that is used during the encryption process. Later, the application can perform a cryptographic function on the encrypted value to determine the underlying data's ordinal value.

Although this is safer than decrypting every single data value, it's not any faster. Each row must be manipulated in order to determine which rows fall within a query's range.

Conversely, binary comparators are generated once for each new data element, and remain static until the tree is refactored, or until the corresponding data element is deleted.

7 Conclusion

Field-level encryption offers the strongest level of encryption, but creates challenges when performing ranged searches.

Because comparators are left-justified within an integer bit field, they maintain an ordinal relationship to the underlying data without leaking information.

Comparators, can be used to perform ranged queries without having to decrypt the underlying data, because they have the same ordinal relationship as the underlying data.

Scaling can be accomplished using integer chaining or strings.

As with data masking, if comparators are suitably unique, the comparator itself can be substituted for the underlying cleartext data value.

Because comparators are ordered, databases can perform ranged searches with optimum performance, because the sort order is preserved. Only the application sees the data in cleartext.

Similar to a database index, periodic maintenance should consist of refactoring the b-tree in order to ensure optimum bit depth and performance.

Unlike other ranged-search schemes, comparators don’t rely on key generation mechanisms, key distribution schemes, nor any fixed relationship with the underlying data.