Security Analytics Core Database Guide, version 10.3

Introduction

The Security Analytics Core products contain a proprietary database developed specifically for use within the Security Analytics suite of products. It bears very little resemblance to traditional relational databases, and it is not based on any off-the-shelf database technology. As such, many users find that there is a steep learning curve to understanding how the Core database works, and how to make best use of it. The purpose of this guide is to help Security Analytics users understand the database and use it to it's fullest potential.

Intended Audience

This guide is intended for multiple audiences. For example, as a System Administrator you can use the information here to help plan your SA deployment, and to tune it for best performance. As a user you can use this guide to structure your analysis in ways that will return reports faster. As a content developer, you can use this guide to help write content that will be processed efficiently by the database system.

SA Products covered by this guide

This guide has been written to cover the capabilities of Security Analytics 10.3. These components of Security Analytics contain the Core Database:

1. Concentrator
2. Archiver
3. Decoder
4. Log Decoder

Terms used throughout this document

Throughout this document you may find the following terms used frequently. Definitions are presented here. The items presented below are ordered by how they enter the Security Analytics system.

- Packet DB: The packet database contains raw packet information. On a Decoder, the packet database contains raw packets as captured from the network. Log Decoders utilize the packet database to store raw log information. Each packet stored in the packet database is accessible by a Packet ID.
- Packet ID: A number used to uniquely identify a packet or raw log in a packet database.

- Meta DB: The meta database contains items of information that are extracted by a Decoder or Log Decoder from the raw data stream. Meta items can generated by parsers, rules or feeds.
- Meta ID: A number used to uniquely identify a meta item in the meta database.
- Meta Key: A name used to classify the type of each meta data. Common meta keys include ip.src, session time, or service type.
- Meta Value: Each meta data contains a value. The value is what is generated by each parser, feed, or rule.
- Session DB: The session database contains information that ties the packet and meta items together into sessions.
- Session: On a packet Decoder, a session represents a single logical network stream. For example, a TCP/IP connection is one session. On a Log Decoder, each log event is one session. Each session contains the references to all the Packet IDs and Meta IDs that refer to the session.
- Session ID: A number used to uniquely identify sessions in the Session DB
- Index: An index is file that provides a way to look-up Session IDs using Meta Values.
- Core Database: This refers to the combination of the Packet, Meta, Session, and Index.

For syntax definitions, this document uses EBNF grammar definitions.

History of Security Analytics Core Database

The SA Core database was developed by NetWitness for use in packet capture systems. Early in the history of NetWitness, we identified that existing database technologies would not be able to keep up with the high ingest rate inherent in full packet capture. Contemporary database technologies were not anywhere close to being able to keep up with capturing the number of sessions received every second, much less sorting every packet. Likewise, the volume of data meant that packet storage would need to be discarded and reused just as quickly as it was consumed. This was also a weakness of databases at the time. Thus, we created a database consisting of the packet, session, and meta databases.

In order to provide the analytical capabilities of NetWitness Investigator, a meta index was added to the NetWitness database. The index shared the same design goals as the original databases: it is designed to sustain a very high insert rate into a high number of very large indices.

The index has evolved considerably over the years. Early versions of the index were only capable of providing summary estimates about how may unique meta values were present in the meta database. Other versions have had great challenges in meeting acceptable query performance. For example version 9.0 of NetWitness more frequently measured report times in

minutes rather than seconds. The current version of the index is derived from the 9.0 NetWitness index, but has evolved considerably in order to meet performance expectations and to add new features.

Core Database Strengths and Weaknesses

Strengths:

- High sustained insert rates, without needing down time for bulk inserts
- Decent query performance simultaneous with high insert rates
- Automatic cleanup/rollover of old data with minimal fragmentation
- Extremely high number of meta value indices: more than 100 enabled by default on a concentrator.
- Ability to scale to Petabyte database sizes and Terabyte index sizes within a single node.
- Using meta key-value pairs, it is very flexible for storing arbitrary meta data within a session. Thus a session can be used to represent nearly any kind of data record.

Weaknesses:

- The query functionality is limited and very low-level.
- The packet/meta/session DB schema is fixed, and all customization is done through custom meta keys and values.
- The database provides no transaction atomicity guarantees as you might expect to find in a SQL database.

Basic Database Configuration

This document assumes that the reader has some familiarity with adjusting the configuration of an SA Core service. The reader should be familiar with one of the mechanisms for modifying the configuration tree of Core services. Examples of such mechanisms include the Explorer view of the Administration pages within the SA user interface, or the REST interface accessible on each service through a web browser.

Finding help within the Core service

Each configuration item within a Core service has a built-in help description of what the item does. This help information can be viewed by hovering your mouse over the configuration item in the Explorer view. Each configuration item also indicates whether it can be changed without restarting or if a restart of the service is needed for the change to take effect.

Developers using the REST API will find they can retrieve the help text for each configuration item by sending the "help" message to the configuration node path.

Packet, Meta, and Session Storage

Each of the packet, meta, and session databases are configured through the /database/config folder on each SA Core service. Each database has a configurable parameter to specify where the Core service will store data. Packet, meta, and session databases follow a predictable pattern for all of their configuration entries. Configuration items for the packet database start with the prefix "packet", meta database configuration starts with the prefix "meta", and the session database configuration items start with the prefix "session".

Index Storage

The index configuration is stored in the /index/config folder on each Core service.

Advanced Database Configuration

The configuration options of the SA Core database may change from one release to the next. However, many of the configuration items do not change frequently and are documented here. This is not an exhaustive list, since new features are added in every release, and they may require new configuration items. For the most up-to-date documentation, refer to the built-in help functionality of the SA Core service.

Description of each database configuration node

packet.dir, meta.dir, session.dir

This is the primary configuration entry for each database. It controls where in the filesystem the database is stored. This configuration entry understands a complex syntax for specifying many directories as storage locations.

Configuration syntax:

```
config-value = directory, { ";" , directory } ;
directory    = path, [ ( "=" | "==" ) , size ] ;
path         = ? linux filesystem path ? ;
size         = number size_unit ;
size_unit    = "t" | "TB" | "g" | "GB" | "m" | "MB" ;
number       = ? decimal number ? ;
```

Example:

```
/var/netwitness/decoder/packetdb=10 t;/var/netwitness/decoder0/packetdb=20.5 t
```

The size values are optional. If set they indicate the maximum total size of files stored there before databases roll over. If the size is not present the database will not automatically roll over, but it's size can be managed using other mechanisms.

The use of `=` or `==` is significant. The default behavior of the databases is to automatically create directories specified when the Core service starts. However, this behavior can be overridden by using the `==` syntax. If `==` is used, the service will not create any directories. If the directories do not exist when the service starts, the service will not successfully start processing. This gives the service resilience against filesystems that are missing or unmounted when the appliance boots.

packet.file.size, meta.file.size, session.file.size

This controls the size of the files created with each database.

packet.files, meta.files, session.files

This setting controls the number of files held open by the database. This value can be increased to improve performance, however the operating system has an overall limit on the number of files that service can keep open. If this limit is exceeded, an error will be reported and the service will not function.

packet.free.space.min, meta.free.space.min, session.free.space.min

This setting provides a safety limit on the minimum free space that exists on the paths specified by the packet.dir, meta.dir, and session.dir directories, respectively. This setting is used to prevent the service from running out of space in the event that other programs have filled up the space that should be dedicated to each of the databases.

packet.index.fidelity, meta.index.fidelity

This setting controls how frequently packet ID locations and meta ID locations are indexed. This setting can be increased to reduce the amount of space needed by each packet or meta nwindex file, but increasing the setting reduces the speed at which individual packets or meta items can be located.

The session database does not have a fidelity setting because it does not generate ID index files.

packet.integrity.flush, meta.integrity.flush, session.integrety.flush

This setting controls whether the database forces a sync operation on the filesystem when it is done writing a file. The default value is true, which means when a file is closed there will be a significant delay while the data is written to non-volatile storage. It may be necessary to set this to false in order to achieve higher sustained write rates.

packet.write.block.size, meta.write.block.size, session.write.block.size

The block size represents how much data is allocated at a time within each database file. Larger block sizes can provide higher throughput and compression ratios, and can improve the rate at which items can be retrieved from the database sequentially. However, larger block sizes have a detrimental impact on random read speed for packet and meta items.

packet.compression, meta.compression

These parameters control whether the databases will compress data. Compression reduces the amount of storage needed by each database, but it can have a major detrimental impact on the speed at which items are written to the database, and the speed at which items are retrieved from the database.

As of 10.3, the valid values for this parameter are `gzip` or `none`.

Descriptions for each index configuration node

index.dir

The index.dir setting controls where the files used by the index are stored. This parameter supports the same syntax as the packet.dir, meta.dir, and session.dir parameters.

page.compression

Deprecated. Versions of the SA Core index between 9.8 and 10.2 supported two different index compression algorithms, and you can choose between them using this setting. As of 10.3 the only recommended value is the default of `huffhybrid`.

SDK Configuration Items that affect Database

There are some additional configuration items in each core service that affect the database, but do not actually affect how the database stores or retrieves data. These settings exist in the `/sdk/config` folder.

max.concurrent.queries

This setting controls how many query operations are allowed on the database simultaneously. Allowing more simultaneous query operations can improve overall responsiveness for more users, but if the query load of the Core device is very I/O bound, having a high max.concurrent.queries value can have a detrimental effect. The recommended value is the number of cores on the system, including hyper threading. Thus, for a Series 4 appliance with 16 cores, the value should be 32.

max.pending.queries

This setting controls the backlog size for the database's query engine. Larger values allow the database to queue more operations for execution. A queued query does not make progress on it's execution, so it may be more useful to make the system produce errors when the queue is full, rather than allowing the queue to grow very large. However, on a system that is primarily performing batch operations such as reports, there may be no detrimental effect to having a large queue.

cache.window.minutes

This setting controls a feature of the query engine that is intended to improve query responsiveness when there are a large number of simultaneous users. See the section on Cache Window under Optimization Techniques.

max.where.clause.cache

The where clause cache controls how much memory can be consumed by query operations that need to produce a large temporary data set to evaluate sorting or counting. If the where clause cache size is overflowed, the query will still work, but it will be much slower. If the where clause cache is too large, it is possible for queries to allocate so much memory that the service would be forced into swap or will run out of memory. Thus, this value multiplied by the max.concurrent.queries should always be much less than the size of physical RAM. This setting understands sizes in the form of a number followed by a unit, for example `1.5 GB`.

query.level.1.minutes, query.level.2.minutes, query.level.3.minutes

The Core database supports three query priority levels. Each core user is assigned to one of the priority levels. Thus, there are up to 3 groups of users that can be defined for the purposes of performance tuning. These setting controls how long each user level is allowed to execute the queries. For example, lower privileged users may have a lower value so that they are not able to use all the Core device's resources with long-running queries.

Per-user Configuration

There are settings that influence the actions users are allowed to perform on the database. These settings are stored in the configuration tree at /users/accounts/*username*/config, where *username* is the name of the user to which the settings apply.

query.prefix

A query prefix applies a filter to every query operation that the user performs. This is implemented by taking the query.prefix values and appending it the the where clause of each query using the logical && (and) operator. See the section on where clauses in this document for

more details.

query.level

The query.level setting assigns the query level that the user will have for every query they perform. These influence whether their queries are limited by the query.level.1.minutes, query.level.2.minutes, or query.level.3.minutes.

session.threshold

The session.threshold setting assigns a maximum session threshold for the user. If set, this threshold value is assigned to all values calls that the user performs. A detailed discussion of both the values call and thresholds is covered in this document.

Rollover

The database operates as a first-in, first-out (FIFO) queue. New data is always appended to the database, and the oldest data is automatically removed as needed. Data that is in the middle of the database is immutable, meaning it cannot be modified.

There are two mechanisms to for rollover: synchronous and asynchronous.

Synchronous Rollover

Synchronous rollover refers to rollover settings that are applied in response to a write operation on the database. That means data will be removed from the database in direct response to the need to write new data. Synchronous rollover is configured by setting size values on the configuration for packet.dir, meta.dir, session.dir, and index.dir.

Synchronous rollover on the packet, meta, and session databases can occur within any write operation.

Synchronous rollover on the index occurs when the index is saved.

Asynchronous Rollover

Asynchronous rollover refers to database file removal that occurs when an explicit rollover command is issued to the database. Most commonly this type of rollover is scheduled to run periodically using the Core service's built-in scheduler. It can also be explicitly requested by the user.

The asynchronous rollover command is the `sizeRoll` message present on the `/index` and `/database` nodes of the configuration tree. The message on the `/database` node will do size rollover on packet meta, and session databases only, while the message on the `/index` node can do simultaneous rollover on both the index and the packet, meta, and session databases. Here is the parameter syntax for the sizeRoll command:

```
size-roll-params  = {type-param, space}, (max-size-param | min-free-param | max-
percent-param)
type-param        = "type=", {type-flag} , { ",", type-flag } ;
type-flag         = "packet" | "meta" | "session" ;
max-size-param    = "maxSize=", number, {space}, unit ;
max-percent-param = "maxPercent=", number, {space}, unit ;
min-free-param    = "minFree=", number, {space}, unit ;
unit              = "t" | "TB" | "g" | "GB" | "m" | "MB" ;
number            = ? decimal number ? ;
percentage        = ? number between 0 and 100 ? ;
```

The `type` parameter controls the databases to consider for removing the oldest data based on total size or space remaining. If type is not specified on the `/index` sizeRoll, only the index is considered for rollover operations.

The `maxSize` sets a current maximum size of the database or index. If the database is larger than this size, oldest data is deleted first until total size is less than maxSize. The sizeRoll operation determines which data is oldest out of all the databases and the index based on session IDs. Sessions or index entries with lowest session IDs are deleted first, possibly including removing meta and packet databases that are orphaned by removing entries from the session database. The index data is rolled out if the sessions that it refers to are removed.

The `maxPercent` parameter sets a maximum percentage of all the volumes of all databases passed in `type` parameter combined. When exceeded, oldest data is deleted first until total size is less than maxPercent of total volumes.

The `minFree` parameter sets a minimum allowed free space on the volumes before oldest data is deleted.

Each call to the sizeRoll operation provides a single pass through the database to delete files. When the operation completes, the database's current size utilization will have met the criteria specified by the `maxSize`, `maxPercent`, or `minFree` parameters. Thus, this operation can be scheduled periodically to ensure that the database can continue to operate uninterrupted.

As an example, here is a typical sizeRoll scheduler entry for an archiver:

```
pathname=/index minutes=5 msg=sizeRoll params="type=meta,session,packet
maxSize=25TB"
```

This scheduler entry specifies that every 5 minutes, the database will ensure that the max size of the meta, session, packet, and index does not exceed 25 terabytes.

Queries

This section covers the query syntax. There are two main mechanisms for performing queries in the database, the `query` call and the `values` on the `/sdk` folder on each Core device.

The `query` call returns meta items from the meta database, possibly using the index for fast retrieval.

The `values` call returns groups of unique meta values sorted by some criteria. It is optimized to return a subset of the unique values sorted by a aggregate function such as count.

Query syntax

The query message has the following syntax:

```
query-params     = size-param, space, query-param, {space, start-meta-param},
{space, end-meta-param};
size-param       = "size=", ? integer between 1 and 1,677,721 ? ;
query-param      = "query=", query-string ;
start-meta-param = "id1=", metaid ;
end-meta-param   = "id2=", metaid ;
metaid           = ? any meta ID from the meta database ? ;
```

The `id1`, `id2`, and `size` parameters form a paging mechanism for the returning a large number of results from the database. Their usage is mostly of benefit to developers who are writing applications directly against the SA Core database.

The `query` parameter is a query command string with it's own SA-specific syntax:

```
query-string      = "select ",  ( "*" | meta-or-aggregate {, meta-or-aggregate}
), " where ",  { where-clause } ;
meta-or-aggregate = meta_key | aggregate_func, "(", meta_key, ")" ;
aggregate_func    = "sum" | "count" | "min" | "max" | "avg" | "distinct" |
"first" | "last" | "len" ;
```

The select clause allows you to specify either `*` to return all the meta in all the sessions that match the where clause, or a set of meta field names and aggregate functions to select a subset of the meta with each session.

The aggregate functions have the following effect on the query result set:

Function	Result
sum	Add all meta values together; only works on numbers
count	The total number of meta fields that would have been returned
min	The minimum value seen
max	The maximum value seen
avg	The average value for the number
distinct	Returns a list of all unique values seen
first	Returns the first value seen
last	Returns the last value seen
len	Converts all field values to a UInt32 length instead of returning the actual value. This length is the number of bytes to store the actual value, not the length of the structure stored in the meta database. For instance, the word "NetWitness" would return a length of 10. All IPv4 fields, like ip.src, would return 4 bytes.

Where clauses

The where clause is a filter specification that allows you to select sessions out of the collection by using the index.

Syntax:

```
where-clause      = criteria-or-group, { space, logical-op, space, criteria-or-
group } ;
criteria-or-group = criteria | group ;
criteria          = meta-key, ( unary-op | binary-op meta-value-ranges ) ;
group             = ["!"], "(" where-clause ")" ;
logical-op        = "&&" | "||" ;
unary-op          = "exists" | "!exists" ;
binary-op         = "=" | "!=" | "<" | ">" | ">=" | "<=" | "begins" | "contains"
| "ends" | "regex" ;
meta-value-ranges = meta-value-range, { ",", meta-value-range } ;
meta-value-range  = (meta-value | "l" ), [ "-", ( meta-value | "u" ) ] ;
meta-value        = number | ( '"' text '"' ) | ip-address | mac-address | ( '"'
date-time '"' ) ;
```

When specifying rule criteria, the meta-value part of the clause is expected to match the type of the meta specified by the meta-key. For example, if the key is `ip.src` the meta-value should be an IPv4 address.

Query Operators

Operator	Function
=	Match sessions containing the meta value exactly. If a range of values is specified, any of the values is considered a match.
!=	Matches all sessions that would not match the same clause as if it were written with the = operator.
<	For numeric values, matches sessions containing meta with the numeric value less than the right side. If the right side is a range, the first value in the range is considered. If multiple ranges are specified, the behavior is undefined. For text metas, a lexicographical comparison is performed.
<=	Same behavior as <, but sessions containing meta that equals the value exactly are also considered matches.
>	Similar to the < operator, but matches sessions where the numeric value is greater than the right side. If the right side is a range, the last value in the range is considered for the comparison.
>=	Same behavior as >, but sessions containing meta that equals the value exactly are also considered matches.
begins	Matches sessions that contain text meta value that starts with the same characters as the right side.
ends	Matches sessions that contain text meta that ends with the same characters as the right side
contains	Matches sessions that contain text meta that contains the substring given on the right side.
regex	Matches sessions that contain text meta that matches the regex given on the right side. The regex is parsing is handled by boost::regex.
exists	Matches sessions that contain any meta value with the given meta key
!exists	Matches sessions that do not contain any meta value with the given meta key

Text Values

Text values are expected to be quoted. Although unquoted strings may work, the values expressed may be ambiguously interpreted as numbers or dates.

It is also imporant to quote any text value may contain `-` so that it is not interpreted as a range.

IP Addresses

IP addresses can be expressed using standard text representations for IPv4 and IPv6 addresses. In addition, the query can use CIDR notation to express a range of addresses. If CIDR notation is used, it is expanded to the equivalent value range.

MAC Addresses

A MAC address can be specified using standard MAC address notation: `aa:bb:cc:dd:ee:ff`

Date and Time expressions

In SA Core dates are represented using Unix epoch time: the number of seconds since Jan 1, 1970 UTC. In queries you can express the time as such a number of seconds, or you can use the string representation. The string representation for the date and time is `YYYY-mmm-DD HH:MM:SS`. The month is represented as a 3 letter abbreviation. Month call also be expressed as a 2 digit number, 01 - 12.

All times specified in queries are expected to be in UTC.

Special Range values

Ranges are normally expressed with the syntax `"smallest" - "largest"`, but there are some special placeholder values you can use in range expressions. You can use the letter `l` to represent the lower-bound of the all meta values as the start of the range, and `u` to represent the upper bound. The bounds are determined by looking at the smallest or largest meta value found in the index out of all the meta values that have entered the index thus far. Note if you use the `l` or `u` tag it should be unquoted.

For example, the expression `time = "2014-may-20 11:57:00" - u` would match all time from that 2014-may-20 11:57:00 to the most recent time found in the collection.

Notice that it is easy to confuse a range expression with a text string. Make sure that text values that contain `-` are quoted, and than hyphens within range expressions are not within quoted text.

Values call

The index provides a low-level `values` function to access the unique meta values that have been stored in the index. This function allows developers to perform more advanced operations on groups of unique meta values.

Values call parameter syntax:

```
values-params    = field-name-param, space, where-param, space, size-param,
{space, flags-param} {space, start-meta-param}, {space, end-meta-param}, {space,
threshold-param} ;
field-name-param = "fieldName=", meta-key ;
where-param      = "where=", where-clause ;
size-param       = "size=", ? integer between 1 and 1,677,721 ? ;
start-meta-param = ? same as query message ?
end-meta-param   = ? same as query message ?
flags-param      = "flags=", {values-flag, {"," values-flag} } ;
values-flag      = "sessions" | "size" | "packets" | "sort-total" | "sort-value"
| "order-ascending" | "order-descending" ;
threshold-flag   = "threshold=", ? non-negative integer ? ;
```

The values call provides the function of returning a set of unique meta values for a given meta key. For each unique value, the values call can provide an aggregate total count. The function used to generate the total is controlled by the flags parameter.

Parameters

Parameter	Function
fieldName	This is the meta key name for which you will retrieve unique values. For example, if fieldName is ip.src, this function will return the unique source IP values in the collection
where	This is a where clause which shall filter the set of sessions for which the unique values are returned. For example, if the fieldName is ip.src, and the where clause is ip.src = 192.168.0.0/16, only values in the range of 192.168.0.0 to 192.168.255.255 will be returned. The where clause syntax is documented in the where clause section of this document.
size	The size of the set of unique values to return. This function is optimized to return a small subset of the possible unique values in the database.
id1, id2	These optional parameters limit the scope of the search for unique values to a specific region of the meta database and the index. Setting the id1 and id2 parameters to a limited range of the meta database is very important to running searches quickly on large collections.
flags	Flags control how the values are sorted and totaled. Flags are documented below.
threshold	Setting the threshold parameter allows the values call to short-cut collection of the total associated with each value once the threshold is reached. By providing a threshold, the caller can reduce the amount of index and meta items that must be retrieved from the database. If the threshold parameter is omitted or set to 0, this optimization is not used.

Values flags

The flags parameter controls how the values call operates. There are three groups of flags that correspond to the different modes of operation:

Flag	Description
sessions, size, packets	The values call allows you to specify one of these flags to determine how the total for each value is calculated. If the flag is sessions, the values call returns a count of sessions that contain each value. If the flag is size, the values call totals the size of all sessions that contain each unique value, and reports the total size for each unique value. If the flag is packets, the the values call totals the number of packets in all sessions that contain each unique value, and then reports that total for each unique value.
sort-total, sort-value	These flags control how results are sorted. If the flag is sort-total, the result set is sorted in order of the totals collected. If the flag is sort-value, the results are returned in order of the sorting order of the values.
order-ascending, order-descending	These flags control the sort order of the result set. For example, if sorting by total in descending order, the values with the greatest total are returned first

Values Call Example

The values call is used extensively by the Navigation view in SA. The default view generates calls that look like this:

```
/sdk/values id1=198564099173 id2=1542925695937 size=20 flags=sessions,sort-
total,order-descending threshold=100000 fieldName=ip.src where="time=\"2014-May-
20 13:12:00\"-\"2014-May-21 13:11:59\""
```

In this example, the navigation view is requesting unique values for ip.src. It is requesting unique values of ip.src in the time range given. It is asking for the count of sessions that match each ip.src, and it the results will be the top 20 ip.src values when sorted by the number total count of sessions in descending order. In addition, the navigation view is providing a meta ID range in order to provide an optimization hint to the query engine.

Stored Procedures

The query and values calls provide more low level search functionality. For more advanced uses cases, server-side stored procedures exist. For more information on the stored procedure API, please refer to the RSA engineering wiki on the topic of Lua Stored Procedure.

Index Customization

Each SA Core service is installed with a default index configuration that is intended to cover the

index needs for most users of the product. However, it is possible to index new meta keys in order to utilize the index with custom content that generated custom meta.

Index configuration file locations

The index customization is accomplished by making changes to the custom index file. The location of this file is /etc/netwitness/ng/index-*servicename*-custom.xml, where *servicename* corresponds to the name of the product that you are customizing. For example, the concentrator custom index file is /etc/netwitness/ng/index-concentrator-custom.xml.

Concentrator products also include a file that describes the default index configuration: /etc/netwitness/ng/index-concentrator.xml. This file is a useful as a template to show how the custom index file is formatted.

If you make customizations to the index in the custom index file, those customizations override any conflict with the default index configuration.

Changes to the custom index file can be made while the service is running. When the service receives an index save command, the changes to the custom index file are read and applied to the index. Note that changes to the index can only be applied to new incoming data. Data cannot be retroactively reindexed with a new custom index configuration, except with a very time consuming reindex procedure.

Index configuration entries

The custom index file is an XML document. The root element of this document is the `language` element, and inside there is elements per meta key to describe each custom index. Each element of the custom index configuration looks like this:

```
<key name="did" description="Decoder Source" level="IndexValues" format="Text"
valueMax="100" />
```

Definitions for each attribute in this element: Attribute | Description -|- name | The name of the meta key that will be indexed description | A human-readable description for the meta type level | The type of index that will be created for this meta key valueMax | The maximum unique values that will be stored for this key per slice format | The format of the data held by all meta items with this meta key name.

In the next few sections we will examine these parameters in greater detail.

Meta names

The meta name used by the index refers to the meta key name present within every meta item in the meta database. These meta names are generated by the Decoders when parsing. Parsers can choose to generate meta with any meta key name. Thus, the custom index allows you to

choose which of the meta items generated by the Decoder are indexed.

Meta key names can be 16 characters long, and contain only letters or the '.' character.

Data Types

When the decoder generated meta items, it assigns a data type to the meta data. Each parser can choose the data type of the meta it generates. However, there are recommended and standard data types for each of the default meta keys. For example, ip.src and ip.dst are stored as the IPv4 meta type, and alias.host is stored as the Text meta type. It is required that each parser agree on the data format for each meta key generated by the Decoder.

When adding it a custom index to the concentrator, it is required that the data type of the custom index match the format of the data generated by the Decoder. If the types do not match the Concentrator will attempt conversions of the meta generated into the type specified for the custom index. However, these conversions sometimes fail, and the resulting index can produce undefined results.

Likewise, when many Decoders and Concentrators are working together as part of an SA installation, it is required that they all agree on the types for each meta key. Conflicts of meta types between SA Core devices can lead to undefined behavior.

Meta data types supported by SA Core

Type	Size in bytes	Description
Int8	1	Signed 8-bit integer
UInt8	1	Unsigned 8-bit integer
Int16	2	Signed 16-bit integer
UInt16	2	Unsigned 16-bit integer
Int32	4	Signed 32-bit integer
UInt32	4	Unsigned 32-bit integer
Int64	8	Signed 64-bit integer
UInt64	8	Unsigned 64-bit integer
UInt128	16	Unsigned 128-bit integer
Float32	4	32-bit floating point number, single precision
Float64	8	64-bit floating point number, double precision
TimeT	8	Unix epoc timestamp
Binary	1-255	Arbitrary binary data
Text	1-255	UTF-8 Encoded text data
IPv4	4	IPv4 address bytes
IPv6	16	IPv6 address bytes
MAC	6	MAC Address bytes

When defining a custom index, it is important to use the best data type for the meta. For example, never store IP addresses as Text, since the Text representation will take more bytes than the IPv4 representation.

Index Levels

There are three levels, or types, of indexing: IndexNone, IndexKeys, and IndexValues.

IndexNone

This type of custom index is not really an index at all. Custom index entries with IndexNone level exist only to define and document the meta key. IndexNone entries can be used in custom Decoder indices to enforce a specific data type for a meta key across all the parsers on a Decoder.

IndexKeys

This type of custom index is indicates that the index will only keep track of sessions that contain meta items with this meta key name. However it will not index any unique values in the meta database for the meta key.

Key-level indices take much less storage space, memory, and CPU time to manage, but they require a lot more work from the query engine when you perform query or values operations using them.

If used in a where clause, a meta key indexed at the key level can only be used to resolve operations such as exists or !exists.

IndexValues

This type of custom index keeps sessions that contain each individual unique value for the meta key. This type of index is also known as a "full index".

This type of index is needed for efficient processing of most where clauses, and for use of this meta key as the fieldName parameter of a values call.

Value Max

Value Max is a parameter that can have a very significant impact on the accuracy and performance of a Value-level index.

As a decoder is parsing packet or logs, it is allowed to create meta of any type with any value. Usually, these meta items are created from data copied directly out of the packet or log. Thus, unique meta values can be created by anyone in response to nearly any event.

The performance of the index is directly dependent on the number of unique values it has found for each meta key. Thus, as the number of unique values goes up, the rate at which new meta is indexed can go down, and the speed with which queries are completed goes down. Thus, since any person can influence the creation unique meta values, it is possible for any person to affect the performance of the index.

The value max parameter limits the number of unique values that can enter the index. Thus, a malicious user cannot flood the system with a large number of unique values in an attempt to make the SA system not work.

It is important to set a value max on any meta key that may have it's value influenced directly by incoming packets or logs.

The value max is applies only to values added since the last index save operation.

The limit for how high value max can be set varies from version to version and on the amount of RAM available to the Core device. As of 10.3, the recommended ceiling for valueMax is 5,000,000 for any meta key. If there are a lot of custom indices, then the valueMax may have to be lower.

Thresholds

Thresholds are a useful optimization that can have a dramatic effect on how fast results are returned to the SA Navigation tool. Thresholds are applied to the values call, as discussed in the values call section of this guide.

The threshold defines a limit to how much of the database is retrieved from disk in order to produce a count. For most queries, the number of sessions that will match the where clause will be very large. For example, selecting all the log events for just 1 hour running at 30,000 events per second would match 108,000,000 sessions.

The threshold feature was introduced based on the observation that most cases where a count of sessions is required do not have to have results that are accurate down to the very last session. For example, when looking at the top 20 IP addresses present over the past hour, it is not terribly important if the report indicates that an IP value matched 10,000,000 or 10,000,001 sessions exactly. The estimate is good enough. In these scenarios, we can make an estimate for the value of the count returned when our count exceeds the threshold parameter. When the threshold is reached, the remaining count is estimated, and the results are sorted based on the estimated counts, if necessary.

Optimization Techniques

The SA Core database is set up to work with a wide variety of work loads by default. However, like any database technology it's performance can be very sensitive to both the nature of the data being ingested, and the nature of the searches that user performs against the database.

Complex where clauses

The amount of time it takes for the SA Core database to produce a result is dependent on the complexity of the query. Queries that align directly with the indexes present on the meta can be resolved quickly, but it is very easy to write queries that cannot be resolved quickly. Sometimes, queries that cannot be returned quickly can be processed by the core database and the index differently to produce much more satisfying results for the customer.

It is useful to know the relative "cost" of each part of the where clause. A clause with a high cost takes longer to execute. In the table below, the query operations are ordered in terms of their relative cost, from lowest to highest.

Operation	Cost
exists, !exists	Constant
=, !=	Logarithmic in terms of the number of unique values for the meta key, linear in terms of the number of unique elements that match a range expression.
<, >, <=, >=	Logarithmic in terms of unique value lookup, but more likely to be linear since the expression will match a large range of values
begins, ends, contains	Linear in terms of the number of unique values the meta key
regex	Linear in terms of the number of unique values for the meta key with a high per-value cost

ANDs and ORs

When constructing a where clause, keep in mind that constructing many terms using the AND operator can have a beneficial effect to the performance of a query. Any time that multiple criteria can be used to filter down the set of sessions matching the where clause, there is less work for the query to do. Likewise, each OR clause creates a larger set of sessions to process for each query.

Thus, as a general rule of thumb, the more AND clauses in the query, the faster it will complete, but the more OR clauses in the query, the slower it will complete.

Use case: match a large subnet

It is common for users to construct queries that attempt to include or exclude a class-A subnet. This type of query is common because the user is trying to include or exclude some large portion of their internal network from their investigation.

It is a problem for the query engine to resolve this query using the ip.src or ip.dst indices alone. The issue arrises from the fact that a where clause such as this:

```
ip.src = 10.0.0.0/8
```

Actually must be interpreted as

```
ip.src = 10.0.0.0 || ip.src = 10.0.0.1 || ip.src = 10.0.0.2 || ... || ip.src =
10.255.255.255
```

Thus, the index could have to create a where clause with more than 16 million terms.

The solution to this problem is to use the Decoder to tag common networks of interest using application rules. For example, you could create meta with an application rule that looks like this:

```
name=internal rule="ip.src = 10.0.0.0/8" order=3 alert=network
```

This rule will create meta in the meta key network with the value internal for any ip address in the 10.0.0.0/8 network.

Thus the where clause could be expressed as:

```
network = "internal"
```

Assuming there is a value-level index on the network meta, the index will not have to expand this query into anything more complex, and the sessions matching the desired subnet will be matched very quickly.

Use case: substring matching

Using the operators begins, ends, contains, and regex in a where clause can be very slow if there are a large number of unique values for the meta key. Each of these operators is evaluated independently against each unique value. For example, if the operator is regex, the regex must be run independently against each unique value.

To work around this, the most effective strategy is to reorganize the meta such that the user does not have to use a substring match.

For example, consider if the user is attempting to find the host name within a URL somewhere in the session. They might write a where clause such as:

```
url contains 'www.rsa.com'
```

In such a scenario it's likely that the url meta key contains one unique value for every session that was captured by the decoder, and therefore will have a huge number of unique values. Thus, the contains operation will be slow.

The best approach is to identify the part of meta they are attempting to match, and move the matching into the content parser.

For example, if there is meta being generated for each url, a parser could also break down to

the url into it's constituent components. For example, if the decoder generates url meta with the value `http://www.rsa.com/security_analytics`, it could also generate alias.host meta with the value `www.rsa.com`. Thus queries could be performed using

```
alias.host = 'www.rsa.com'
```

Since the substring operator is no longer needed, the query will be much faster.

Index Saves

The Core index is subdivided by save points, also known as slices. When the index is saved, all the data in the index is flushed to disk, and that portion of the index is marked as read-only. Saves serve two functions:

1. Each save point represents a place where the index could be recovered in the case of a power failure.
2. By periodically saving, we can ensure that the portion of the index that is actively being updated does not grow larger than RAM.

Save points have the effect of partitioning the index into independent, non-overlapping segments. When a query must cross over multiple save points, it must re-execute parts of the query and merge the results together. This ultimately makes the query take longer to complete.

By default a save is performed on the core index every 8 hours. You can see the current save interval by using the scheduler editor in the SA Administration UI for the device. The default entry looks like this:

```
hours=8 pathname=/index msg=save
```

By adjusting the interval we control how often saves are created.

Affects of increasing the save interval

By increasing the save interval, save points are created less frequently, and therefore fewer save points will exist. This has a positive effect on query performance, because it becomes less likely that queries will traverse slices, and when slices do have to be traversed, there are not as many to traverse.

There are downsides to increasing the save interval though. First, the concentrator is more likely to hit the valueMax limit set on any of the indices. Second, the recovery time in the event of a forced shutdown or power failure is increased. And third, the aggregation rate may suffer if the index slice grows too large to fit in memory.

Affects of decreasing the save interval

By decreasing the save interval, it is possible to avoid hitting the valueMax limits while maintaining a full value index for meta that contains a large number of unique values. Decreasing the save interval does have a detrimental impact on query performance, since more slices will be created.

Working with Value Max

The value max limitation can be frustrating to customers that want to index all possible unique meta. Unfortunately that is not possible in the general case. There will exist meta keys that can have arbitrary random data from anywhere on the Internet, and all unique values cannot be indexed.

However, it is possible to work around some of the limitations of value max by utilizing key level indices instead of value indices. Key level indices are not influenced by value max.

It is possible to use the Navigation view on a meta key indexed at the key level. The database will utilize value level indices in the where clause where possible, but meta database scanning will be used to resolve unique values for the values call. This approach will work well when the where clause provides an effective filter to limit search scope to a small number of sessions, perhaps less than 10,000 sessions.

In cases where the value max is reached, the user can perform a database scan on their queries to ensure no relevant values were dropped. This feature is accessible in the 9.8 Investigator client via the right-click menu on the navigation view. Although the meta database scan will take a long time, it can provide reassurance to the customer that they are not missing anything in their reports.

Parallelize Workloads

When the customer is using a lot of reports, ensure that they are making full use of the parallel executing options within Reporting Engine. Likewise, ensure that the number of max.concurrent.queries is appropriate for the hardware.

The Investigator client has the ability to run the components of the Investigator view in parallel, which can have a significant impact on the perceived performance of the SA core device. This feature is expected to reappear in SA UI 10.4.

Index Rebuild

In rare cases, a core device might benefit from an index rebuild. Examples:

1. The SA core device has index slices created by a very old version of the product and has not rolled-out any data in more than 6 months
2. The index was configured incorrectly, and the customer wants to re-index all meta with a

new index configuration
3. The traffic load into the core device was very light, and the save interval was large, causing more slices than needed to be generated.

In these cases, and index rebuild may provide performance improvements. To do so, you must send the message `reset` with the parameter `index=1` to the /decoder folder on a decoder, the /concentrator folder on a concentrator, or the /archiver folder on an archiver.

Be aware that a full reindex will take days to complete on a fully loaded concentrator, and possibly weeks on a full Archiver.

Scaling Retention

There are several ways to improve the retention of the SA Core database. Here retention refers to the period of time which is covered by data stored in the database.

The first step in analyzing retention is to determine which part of the database is the limiting factor in terms of retention. The packet, meta, and session databases provide the packet.oldest.file.time, meta.oldest.file.time, and session.oldest.file.time stats in the /database/stats folder to show how old the oldest file in the database is. The index provides the /index/stats/time.begin stat to show the oldest session time stored in the index.

Increasing Packet and Meta Retention

The primary mechanism for increasing retention on these databases is adding more storage. If adding more storage to the SA Core device is not possible, then it may be necessary to utilize the compression options on the packet and meta database to reduce the amount of data each database writes.

If meta retention is a concern, it may be wise to remove unneeded content from the Decoder generating meta. Many parsers generate meta that the customer does not need to store long term.

Increasing Index Retention

Usually the index has longer retention than the databases, but with a complex custom index the index retention may be shorter. Usually the easiest course of action is to remove unneeded value-level indices from the the custom config, or perhaps override the some of the default value-level indices with key-level indices.

It is also possible to scale the index by adding additional index storage. However, the index storage should be extended using solid-state drives only.

Scaling Horizontally

Starting in version 10.3, Concentrators and Archivers have the ability to be clustered with a

feature known as "gang aggregation." This feature allows a single Decoder to feed sessions to multiple Concentrator or Archivers in a load-balanced fashion.

This feature allows the query and aggregation workload to be split among an arbitrarily large pool of hardware.

For more information on configuring this feature, consult the topic Clustering Concentrator or Gang Aggregation on the RSA engineering wiki.

Grouping Workloads

The SA core database works much better when all the users of the system are working within the same region of the database. Since the database is fed data from the Decoder with a first-in-first-out scheme, data in the database typically is clustered together according to the time it was captured and stored. Therefore, the database works better when all users are working on the same time period of data.

Naturally, it will not always be possible for all users to be working on the same time period simultaneously. The SA Core database can handle that use case, but it will be slow to do so because it must alternate between having different periods of time in RAM. It is not possible to have all of the time periods in RAM at the same time. Typically less than 1% of the database and less than 10% of the index will fit in RAM.

To make SA work for the customer, it is important to get the customer to organize their users into groups that will tend to work on the same time ranges. For example, users who do daily monitoring over the most recent data may be one user group. Perhaps there will be another user group that does queries further back in time as part of an investigation. And perhaps another set of users will be doing reports over large periods of time. Attempting to serve all the groups from a single database can lead to frustration and long wait times for results to be produced. However, if the different use cases can be spread to different concentrator hardware the perceived performance can be much better. In this case it may be beneficial to utilize more Concentrator service with less RAM and CPU power rather than a single large / expensive concentrator intended to meet all needs.

Cache Window

Consider this sequence of events:

1. At 9:00 a.m., user "kevin" logs into a Concentrator and requests a report on the last 1 hour of collection time.
2. The concentrator retrieves reports for the time range 8:00 a.m. to 9:00 a.m.
3. At 9:02 a.m., user "scott" logs into the same Concentrator and also requests a report on the last 1 hour of collection time.
4. The concentrator retrieves reports for the time range 8:02 a.m. to 9:02 a.m.

Notice that even though both users were looking at time ranges that were close together, the

work done by the concentrator to produce Kevin's reports could not be re-sent to Scott, since the time ranges are slightly different. Thus the Concentrator had to re-calculate most of the reports for Scott.

The setting cache.window.minutes on the /sdk node allows you to optimize this situation. When a user logs in, the point in time representing the most recent data for the collection only moves forward in increments of the the number of minutes in this setting.

For example, assume the /sdk/config/cache.window.minutes is 10. If we re-evaluate the action above, we get a new sequence of events.

1. At 9:00 a.m., user "kevin" logs into a Concentrator and requests a report on the last 1 hour of collection time.
2. The concentrator retrieves reports for the time range 8:00 a.m. to 9:00 a.m.
3. At 9:02 a.m., user "scott" logs into the same Concentrator and also requests a report on the last 1 hour of collection time.
4. The concentrator retrieves reports for the time range 8:00 a.m. to 9:00 a.m.
5. At 9:10 a.m., user "scott" re-loads the reports for the last 1 hour of collection time.
6. The concentrator retrieves reports for the time range 8:10 a.m. to 9:10 a.m.

The report returned at step 3 falls in the cache window, so it will be returned instantaneously. This will give Scott the impression that the Concentrator is very fast.

Thus, larger cache.window settings improve perceived performance, at the cost of introducing small delays until the latest data is available to search.

Time Limits

When a query is running on the SA Core database for a very long time, the Core service will dedicate more and more CPU time and RAM to that query in order to get it to complete faster. This can have a detrimental impact on other queries and aggregation. In order to prevent lower privileged users from utilizing more than their share of the Core service's resources, it is a good idea to put time limits on the queries run by normal users.

Appendix: Statistics

The Core services provides a very large number of statistics for monitoring the operation of the system. Some of them are useful for monitoring performance, while some of them exist for monitoring the operation of the system or for debugging purposes.

Database Statistics

Statistics in /database/stats

Statistic	Meaning
meta.bytes, packet.bytes, session.bytes	The total size of data stored in each database
meta.first.id, packet.first.id, session.first.id	The first meta ID, packet ID, and session ID, respectively, stored in the database
meta.last.id, packet.last.id, session.last.id	The last meta ID, packet ID, and session ID, respectively, stored in the database
meta.oldest.file.time, packet.oldest.file.time, session.oldest.file.time	The creation date of the oldest file in each database
meta.rate, packet.rate, session.rate	The count of the number of meta, packet, and session objects added to each database over the last second
meta.total, packet.total, session.total	The total number of meta, packet, and session objects within each database
meta.volume.bytes, packet.volume.bytes, session.volume.bytes	The approximate total volume size for all directories used by each database
meta.free.space, packet.free.space, session.free.space	The approximate total unused space across the all directories used by each database

Statistics in /index/stats

Statistic	Meaning
checkpoint.page, checkpoint.summary	The last objects stored the last time an index save was created. (debugging)
index.bytes	An approximate measure of how much disk space is required by index files
memory.used	An approximate measure of how much memory is occupied by the index
page.first.id, summary.first.id	The first page and summary object stored in the index (debugging)
page.last.id, summary.last.id	The last page and summary object stored in the index (debugging)
page.total, summary.total	Number of pages and summaries in the index (debugging)
session.first.id	The ID of the first session indexed
session.last.id	The ID of the last session indexed
sessions.since.save	The number of sessions currently held by the current index slice
values.added	The number of unique values added to the current index slice
slices.total	The number of slices in the index
time.begin	The oldest time meta indexed
time.end	The most recent time meta indexed

Statisitcs in /sdk/stats

Statistics	Meaning
cache.window.time.begin	The beginning of the current time enforced by cache.window.minutes
cache.window.time.end	The end of the current time enforced by cache.window.minutes
queries.active	The number of queries currently executing in the index
queries.queued	The number of queries waiting for execution

Per-query statistics

SDK operations like query and values provide information about their execution status in

/sdk/config/stats/queries/*handleid*, where *handleid* is a unique identifier for the query operation.

Statistic	Meaning
channel.path	This stat provides a link to the connection channel over which the operation is communicating. This channel is used to commuicate results back to the client.
query.type	The type of operation being performed, such as queries or values
query	The complete set of parameters given to the query
query.progress	The percentage of the query execution that has completed.
query.status	A message describing what stage of the query execuition is currently occurring
running.since	The time at which the query began execution
user	The user name that executed the query