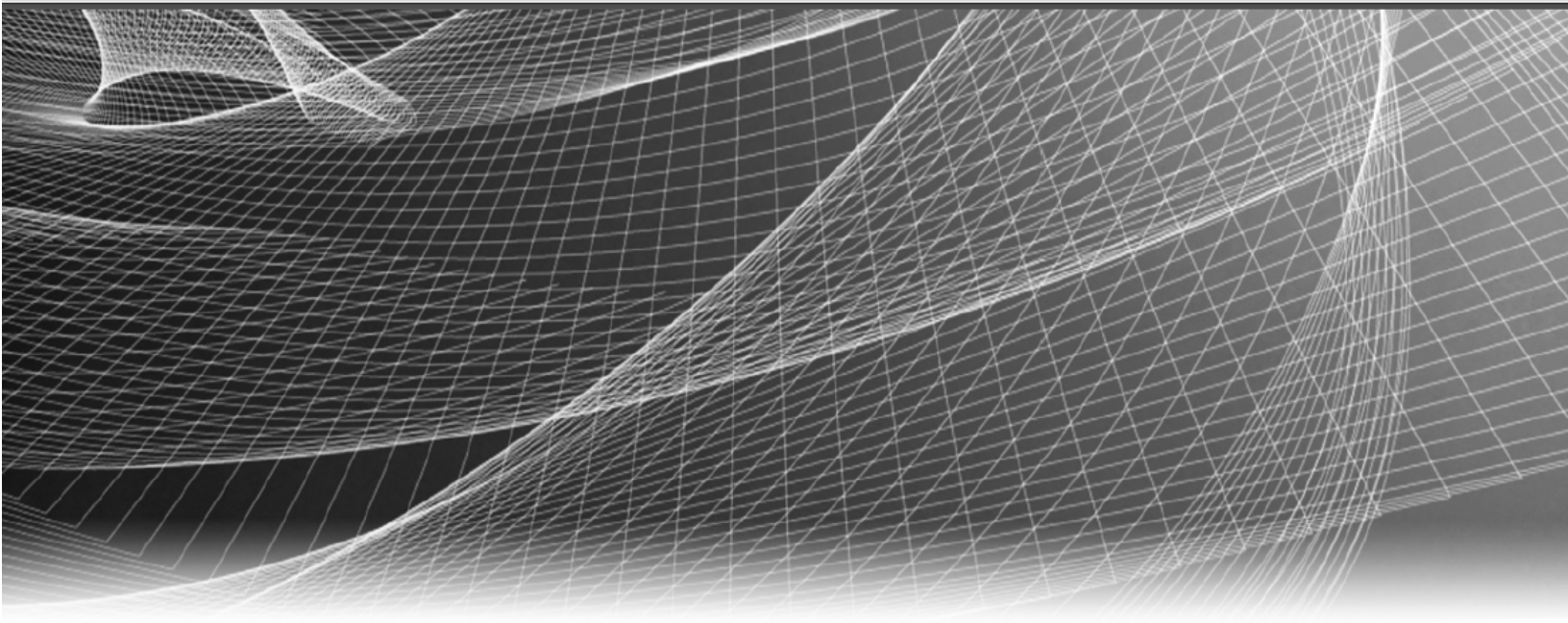

RSA Security Analytics

Core Database Tuning Guide

for Version 10.6.5

RSA

EMC²

Contact Information

RSA Link at https://community.rsa.com contains a knowledgebase that answers common questions and provides solutions to known problems, product documentation, community discussions, and case management.

Trademarks

For a list of RSA trademarks, go to www.emc.com/legal/emc-corporation-trademarks.htm#rsa.

License Agreement

Third-Party Licenses

This product may include software developed by parties other than RSA. The text of the license agreements applicable to third-party software in this product may be viewed on the product documentation page on RSA Link. By using this product, a user of this product agrees to be fully bound by terms of the license agreements.

Note on Encryption Technologies

This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when using, importing or exporting this product.

Distribution

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

October 2017

Contents

Security Analytics Core Database Introduction

This topic provides an overview of the Security Analytics Core database. The Security Analytics Core services contain a proprietary database developed specifically for use within the Security Analytics suite of products. It bears little resemblance to traditional relational databases, and 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 its fullest potential.

As a System Administrator, you can use this information to help plan your Security Analytics deployment, and to tune it for best performance. As an Analyst, 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.

Security Analytics Products Covered by this Guide

This guide covers the capabilities of Security Analytics 10.6. The following Security Analytics components contain the Core database:

- Concentrator
- Archiver
- Decoder
- Log Decoder
- Workbench

Frequently-Used Terms

Definitions for terms that are used throughout this document are presented here. The terms are listed in the order in which they enter the Security Analytics system:

- **Packet DB**: The packet database contains the raw captured data. On a Decoder, the packet database contains packets as captured from the network. Log Decoders use the packet database to store raw logs. The raw data stored in the packet database is accessible by a Packet ID, however, this ID is typically never visible to the end user.
- **Packet ID**: A number used to uniquely identify a packet or 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. Parsers, rules, or feeds can generate meta items.
- **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 item. Common meta keys include ip.src, time, or service.
- **Meta Value**: Each meta item contains a value. The value is what each parser, feed, or rule generates.
- **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**: The index is a collection of files 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.

Security Analytics Core Database History

NetWitness developed the Security Analytics Core database for use in packet capture systems. Early in the history of NetWitness, developers 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, NetWitness 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 was 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 many unique meta values were present in the meta database. Other versions have had great challenges in meeting acceptable query performance. For example, NetWitness 9.0 more frequently measured report times in minutes rather than seconds. The current version of the index is derived from the NetWitness 9.0 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 and 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 items within a session. Thus a session can be used to represent nearly any kind of data record.

Weaknesses:

- The query functionality is limited and low level.
- The packet, meta, and 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 topic covers basic database configuration settings of Security Analytics Core services. For information on how to configure the Core services by editing configuration files, see **Service Configuration Settings** in the *Host and Services Getting Started Guide*.

This document assumes that the reader has some familiarity with adjusting the configuration of a Security Analytics Core service. To use this document, you 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 Security Analytics user interface, or the REST interface accessible on each service through a web browser.

Find Help within the Core Service

Each configuration item within a Core service has a built-in help description of what the item does. You can view this help information 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 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 Security Analytics Core service. Each database has a configurable parameter to specify where the Core service stores 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.

Topics

Tiered Database Storage

This topic describes tiered database storage and provides recommendations for *Hot*, *Warm*, and *Cold* tier storage.

Starting with version 10.4, the Archiver service has the capability to be configured to use tiered storage. The concept of tiered storage is to put the most recent data on a *Hot* tier, which is the fastest storage available on the Archiver.

Note: All services use the *Hot* tier by default.

The next tier is known as *Warm* and is typically cheaper and slower storage, such as a network-attached storage (NAS). The *Warm* tier contains older data; how old depends upon how much storage is allocated on the *Hot* tier and the average ingest rate. When the *Hot* tier reaches max utilization, the natural progression is to move the oldest data from the *Hot* tier to the *Warm* tier. When configured correctly, this happens automatically and is invisible to the end user. Queries and data access happen automatically no matter what tier (*Hot* or *Warm*) the data resides on. However, there can be a performance impact when accessing data on the *Warm* tier as compared to the *Hot* tier, because access times on the *Warm* tier are typically slower.

In addition to *Hot* and *Warm*, there is also a *Cold* tier. The *Cold* tier is only used as a staging area for offline backup. Security Analytics Core services do not access data on the *Cold* tier. Security Analytics Core services move the oldest data to the *Cold* tier and consider it abandoned (the service no longer accesses the data). This data can then be backed up to long-term storage like tape for possible restoration months or even years later, depending on requirements. The backing up and subsequent removal of data on the *Cold* tier must be handled outside of Security Analytics Core services via scripts or other processes.

Caution: If the *Cold* tier becomes full because external processes are not removing data in a timely manner, this causes the Security Analytics Core service to eventually stop the ingestion of new data until the problem is corrected.

When moving data to the *Cold* tier, RSA recommends that the directory remain on the same mount point as where it is being moved from. Therefore, if the files are coming from the *Warm* tier, it is far better for performance reasons to set the *Cold* tier directory on the same file system. The reason for this is that the service attempts to simply move the file and directory to the *Cold* tier, which is a nearly instantaneous operation on the same file system. If the move fails, the fallback is to copy the data to the *Cold* tier, which takes more processing time and causes additional I/O contention on the tier from which it is being copied.

Archiver

The tiers of storage capabilities are used by the Archiver. You can configure Archiver to only use *Hot* storage (the default), *Hot* and *Warm*, or all three (*Hot*, *Warm* and *Cold*). All services must use *Hot*, you cannot configure a service to only use *Warm*. Data flows from *Hot* to *Warm* and finally to *Cold*. You can also skip *Warm* and go from *Hot* to *Cold*. If *Cold* (offline) storage is not configured, the oldest data is deleted on the last configured tier, which has been the standard operating procedure.

The typical Archiver deployment sets all the databases to unlimited size (packet.dir, meta.dir, session.dir, index.dir, and optionally the *Warm* tier variants), which means that the size specifier is left off or set to zero. This lets the databases and index grow unbounded. Instead of each database managing their own size and rolling out only when each individual database exceeds their configured size, Archiver rolls out everything together using the **/index sizeRoll** command. This enables the databases and index to roll out in unison. For more information on sizeRoll, see *Asynchronous Rollover* in Rollover.

Archiver is typically configured to place the index, session, meta, and packet (log) DB on the same volume, instead of multiple volumes like a Concentrator or Decoder. Although this can potentially cause more I/O contention when concurrent reads happen across multiple databases, it also maximizes overall retention. Because all databases are on the same volume, they are configured to roll out together, which minimizes orphaning of data. Decoder and Concentrator are configured for maximum I/O speed, but can suffer from estimates on the proper volume sizing.

For example, if the session DB is too large, it may have enough storage for six months of retention, whereas the meta DB and index only have retention for four months. Because the session, meta DB, and index are intricately tied together, the shortest retention period for all three define the overall retention period (in this case, four months). Retention of individual databases is mostly affected by factors beyond our control, such as traffic captured, meta generated (parsers, feeds, rules) and filtering. The databases are easily resized by a simple configuration change, but this usually also involves changes at the hardware and file system level to adjust partitions, which complicates dynamic resizing. Archiver avoids these problems by using a single volume for everything, with the trade-off of somewhat slower I/O speed.

Manifests

This topic describes manifest files and provides an example manifest for a meta DB file. It also describes manifest searching and provides an example manifest search.

Manifest files are created with every session, meta, and packet (log) DB file and index slice directory. A manifest file is a file that describes several key pieces of information about the data to which it refers. Manifest files are written as a JSON record. Manifest files travel with the data they represent from tier to tier. If the data they represent is deleted, the manifest file is also deleted, except in the following special case. If the service has **/database/config/manifest.dir** configured to a valid directory, at the point when the manifest data is deleted, a copy of the manifest file is placed into the directory pointed at by **manifest.dir** (the directory is created if it does not exist). This enables a Security Analytics feature called historical manifest searching.

The intention of this process is to keep historical manifest files for years, in one location for offline querying. As you might imagine from a service running for many years, this can potentially generate hundreds of thousands of files. This should not be a concern however, as the service automatically compresses files into a single archive in order to save space when they grow too numerous. Manifest files are very small and compress well.

Example manifest (**meta-000000023.nwmdb.manifest**) for a meta DB file:

```
    {
        "filename" : "meta-000000023.nwmdb",
        "size" : 185153768,
        "fileTime" : 1403903940,
        "id1" : 150814110,
        "id2" : 159341086,
        "session1" : 4023382,
        "session2" : 4250442,
        "time1" : 1403903879,
        "time2" : 1404739851
    }


filename  = The filename for the db file the manifest represents
size      = The size in bytes of the db file
fileTime  = The time the file was created
id1       = The starting id in the file (for this example, the starting
meta ID)
```

```
id2       = The last id in the file (for this example, the last meta ID)
session1  = The starting session ID of the first meta in the file
session2  = The last session ID of the last meta in the file
time1     = The POSIX time of the first "time" meta found in the file
time2     = The POSIX time of the last "time" meta found in the file
```

In this example manifest, the most important fields are **fileTime, time1** and **time2**. All three fields are written in POSIX time. **time1** and **time2** are the starting and stopping times of the meta recorded in the meta DB file **meta-000000023.nwmdb**. In particular, **fileTime** is always the time in which the file was created (not last modified). **time1** and **time2** are representative of the min and max range of the parsed data within the meta DB file. When doing historical searches by time, **time1** and **time2** are preferred over **fileTime**, when they are present. Manifest files for the other databases and index contain some different fields, but all have enough information to perform time based queries.

Search Historical Manifests

When manifests are collected in the directory pointed to by **manifest.dir**, it is assumed that the data they refer to was copied to the *Cold* tier and eventually backed up to offline storage. Because the historical manifests are still accessible by the service, this allows time-based queries to be performed on offline data, in order to determine what data needs to be restored for a given time range.

You can search manifests using the **/database manifest** command:

```
manifest: If a manifest directory is defined, it will allow operations
on the manifest files (such as a time based query) for database files in
cold storage.
security.roles: database.manage
parameters:
   op - <string, optional, {enum-one:query|compress}> The operation to
perform (defaults to query)
   time1 - <date-time, optional> The beginning time (UTC) for matching
offline database files
   time2 - <date-time, optional> The ending time (UTC) for matching off-
line database files
   timeFormat - <string, optional, {enum-one:posix|simple}> Specify the
time format that is returned (posix, simple), default is posix
```

Example search:

/database manifest time1="2014-04-20 11:00:00" time2="2014-04-11 11:20:00" timeFormat=simple

The search returns all manifests that match the query:

```
    [ filename=meta-000001691.nwmdb size=4843826176 fileTime="2014-Apr-
20 11:06:34" id1=301555027452 id2=301733101896 session1=15352020201 ses-
sion2=15361024200 time1="2014-Apr-20 11:05:34" time2="2014-Apr-20
11:16:34" compression=gzip ]
    [ filename=session-000001865.nwsdb size=268439552 fileTime="2014-
Apr-20 11:06:35" id1=14674145801 id2=14682041000 metaId1=288217522208
metaId2=288370660984 packetId1=11733872441 packetId2=11741745303 ]
    [ filename=session-000001866.nwsdb size=268439552 fileTime="2014-
Apr-20 11:18:31" id1=14682041001 id2=14689936200 metaId1=288370660985
metaId2=288520616949 packetId1=11741745304 packetId2=11749618589 ]
```

The returned results can be used to correlate which files should be restored from backup for the given time range. For Security Analytics 10.4 and later, a service called Security Analytics Workbench can be used to take the restored files and provide a query interface over the restored data using one or more collections.

Setup of the Workbench service is beyond the scope of this document. For more information, see **Configure Data Backup and Restore** in the *Archiver Configuration Guide*

Advanced Database Configuration

This topic explains the advanced configuration options of the Security Analytics Core database.

The configuration options of the Security Analytics 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 Security Analytics Core service.

Topics

Database Configuration Nodes

This topic describes database configuration nodes. The following database configuration nodes are some of the advanced database configuration items of the Security Analytics Core database that do not change frequently.

packet.dir, meta.dir, session.dir

This is the primary configuration entry for each database (also known as the *Hot* tier). It controls where in the file system the respective databases are 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/net-
witness/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 does not automatically roll over, but its 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 does not create any directories. If the directories do not exist when the service starts, the service does not successfully start processing. This gives the service resilience against file systems that are missing or unmounted when the host boots.

If you modify the size of a directory in use, the size takes effect immediately, as long as it is larger. If the size is smaller, it is ignored if it is more than 10 percent smaller than the existing size. This prevents an accidental mistype that causes a enormous loss of data. For example, if the packet database was configured for `12 TB` and someone mistyped it as `12 GB`, the database would end up deleting over 11 TBs of data in order to shrink it down to just 12 GB. Instead, the database ignores the `12 GB` setting and logs a warning, so that the error can be caught quickly. Of course, if the size specified is actually correct and more than a 10 percent difference from the existing size, the only recourse for it to take effect is to restart the service. When it starts back up, it assumes the size is correct and adjusts the database to the new size by rolling out the oldest data until the new size is reached. If you actually do want to adjust the size downward and by more than 10 percent without restarting the service, you need to modify the size multiple times, each time adjusting it by less than 10 percent. Watch the service logs to know when the database has adjusted to the new size, as it only adjusts the total database size when the latest file being written has been closed.

If new directories get added or deleted (semicolon separated), they do not take effect until the service restarts.

packet.dir.warm, meta.dir.warm, session.dir.warm

These settings are optional and are used for *Warm* tier storage on an Archiver. By default, they are blank and unused. If configured, they follow the same format and behavior as packet.dir, meta.dir, and session.dir (see *packet.dir, meta.dir, and session.dir* above). When configured, the oldest file on the *Hot* tier moves to the *Warm* tier when no available space remains in the *Hot* tier.

packet.dir.cold, meta.dir.cold, session.dir.cold

These settings are optional and are used to move files from either a *Hot* or *Warm* tier storage system to the *Cold* tier directory specified. Specifically, this setting is nothing more than a directory, there are no size specifiers. However, the defined path name has a few special format specifiers that you can use to name the directory with the date of the data in it.

```
%y   = The year of the data being moved to the cold tier
%m   = The month of the data being moved to the cold tier
%d   = The day of the data being moved to the cold tier
%h   = The hour of the data being moved to the cold tier
%##r = A block of time within a day. So %12r would create two blocks, 00
and 01. 00 for all data in the AM, 01 for all PM data
```

Example setting:

```
packet.dir.cold = /var/netwitness/archiver/database1/alldata/cold-stor-
age-%y-%m-%d-%8r
```

For the setting above, if a log database file was about to be moved to cold storage and it was created on `2014-03-02 15:00:00`, it would be moved to the following directory on the *Cold* tier:

```
/var/netwitness/archiver/database1/alldata/cold-storage-2014-03-02-01
```

The last number `01` needs some explanation. The `%8r` specifier breaks the hours of the day into 24 / 8 = 3 parts. The first eight hours of the day would be block `00`, so 12 a.m. to 8 a.m. The next eight hours are from 8 a.m. to 4 p.m. and are assigned block `01`. Since the data being moved to cold storage was created at 3 p.m., it falls into block `01`. The `%r` format specifier is useful for backing up files with a granularity somewhere between a day `%d` and a single hour `%h`. The *Cold* storage directory is created on demand and is defined by the data being moved when the format specifiers are used.

The ability to add a date to the path of the data is just a convenience added for backup and restore. It is a way of tagging the data with a date in the path.

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

This controls the size of the files created with each database. It is normally not necessary to change these values as the default values typically work well. This setting takes effect immediately for subsequent files.

packet.files, meta.files, session.files

This setting controls the number of files held open by the database. You can increase this value 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 is reported and the service does not function. This setting takes effect immediately.

In Security Analytics 10.6 and later, the default value for packet.files, meta.files, and session.files is `auto` and the service manages the number of open files based on this criteria:

1. Number of collections
2. Amount of system memory

When set to `auto`, the number is dynamic and you can view it in the logs when it changes. For Security Analytics 10.6, RSA recommends that you set this value to `auto` and do not change it to a specific number.

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. This setting takes effect immediately.

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. This setting takes effect immediately.

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

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

This setting controls whether the database forces a sync operation on the file system when it is finished writing a file. The default value is `sync`, which means when a file is closed there will be a significant delay while the data writes to non-volatile storage. It may be necessary to set this to `normal` in order to achieve higher sustained write rates, especially on a Decoder. This setting takes effect on the next file created. Therefore, it is expected that at least one more sync will happen if the value was just changed to `normal`.

If packet drops are occurring and packet.integrity.flush is set to `sync`, set it to `normal` and monitor. Keep the session and meta flush settings on `sync`. If packet drops are still problematic, then set all three to `normal` and monitor.

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 *compressed* packet and meta items. This setting takes effect immediately.

packet.compression, meta.compression

These parameters control whether the databases 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. Changes take effect immediately on the next file creation.

As of Security Analytics 10.4, the valid values for this parameter are `gzip`, `bzip2`, `lzma`, or `none`. `gzip` is the preferred algorithm when compression is used, because it provides a good balance between performance and space savings. Both `bzip2` and `lzma` can achieve better space savings, but the tradeoff in speed is substantial and likely should only be considered for low ingest speeds and when storage space is at a premium.

packet.compression.level, meta.compression.level

You can use these settings to further refine how the compression algorithms behave. They have no effect when compression is disabled. The valid values are between 0–9. The default value of zero means let the software pick the best setting for speed and compression. The values between 1 and 9 are used as a sliding scale between performance (1) and compression (9). The value of 9 typically gives you the best compression for a given algorithm, but the worst performance. Somewhere in the middle is usually the best setting, which is what zero picks.

hash.algorithm

This setting controls how the database files are hashed. The default value is `none`, so no hashing is performed. The valid values are `none`, `sha256`, `sha1`, or `md5`. Database files can be hashed to provide evidence that they have not been tampered with since they were closed. Hashing is time intensive and affects ingest performance when enabled. This change takes effect immediately.

hash.databases

This setting controls which databases are hashed. Valid values are `session`, `meta`, and `packet` and are comma separated when hashing multiple databases. This change takes effect immediately.

hash.dir

This setting is normally empty, which means the hash file is created in the same directory as the database file that was hashed. If this setting is defined, the hash file is written to the directory specified instead. This could be some form of write-once storage for resilience against hash tampering.

Hash files are small XML files containing the hex encoded hash along with metadata about the database file that was hashed.

Index Configuration Nodes

This topic describes index configuration nodes. The following index configuration nodes are some of the advanced database configuration items of the Security Analytics Core database that do not change frequently.

index.dir

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

index.dir.warm

The *Warm* tier storage for index slices. This setting supports the same syntax as packet.dir.warm, meta.dir.warm, and session.dir.warm.

index.dir.cold

The *Cold* tier storage for index slices. This setting supports the same syntax as packet.dir.cold, meta.dir.cold, and session.dir.cold.

index.slices.open

This setting controls the number of index slices held open by the index. Index slices are opened automatically as needed by queries. When queries complete, the index engine may hold the slices open so that subsequent queries execute faster. The most recently created slices are the slices that will be held open, since they are mostly likely to be used by queries.

If queries against the index require the index to open slices, then they will execute slower than if the slices were already open. Therefore, this parameter should be tuned such that most queries executed against the index will work on open slices. However, each open index slice consumes some resources, such as file handles and memory. If there are too many index slices open, the overall performance of the service can suffer.

You should set this parameter so that the open index slices will cover most of the time ranges that most queries will need. For example, if most queries are over the past two weeks, and there are index slices created every 8 hours, then there are 14 days x 3 slices per day, or 42 slices created over the past two weeks. Thus, you could set index.slices.open to 42 so that only slices that are likely to be used are held open.

If this parameter is set to 0, then all slices are held open until the next index save. In this scenario, the only thing limiting the number of slices open in the process is the number of slices in the index.

page.compression

Deprecated. Versions of the Security Analytics 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`.

save.session.count

This setting controls how often the index is automatically saved when new sessions are inserted. If the value of save.session.count is greater than 0, any time more than save.session.count sessions are added to the index, the index automatically saves itself. If the save.session.count is set to 0, this feature is disabled and the index will not automatically save itself when new sessions are added to the index.

Save.session.count can be used to implement an automatic save pattern that is based on the volume of data that enters the index. This is useful because it allows a lightly loaded system to generate save points less often.

For more information on the topic of index saves, see the section in this guide on Optimization Techniques.

SDK Configuration Nodes

This topic describes the SDK configuration nodes that affect the 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 service is very I/O bound, having a high max.concurrent.queries value can have a detrimental effect. The recommended value is near the number of cores on the system, including hyper threading. Thus, for an appliance with 16 cores, the value should be somewhere close to 32. Subtract a few for aggregation threads and general system response threads. Subtract a few more if this is a hybrid system (for example, both a Decoder and Concentrator running on the same appliance). There is no magic number, but somewhere between 16 and 32 should work well.

max.pending.queries

This setting controls the backlog size for the query engine of the database. Larger values allow the database to queue more operations for execution. A queued query does not make progress on its 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. For more information on cache window, see 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 still works, but it is 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 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

These settings are available in Security Analytics 10.4 and earlier versions.

In Security Analytics 10.4 and earlier, the Core database supports three query priority levels. Each core user is assigned to one of the priority levels. Therefore, there are up to three groups of users that can be defined for the purposes of performance tuning. These settings control 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 resources of the Core service with long-running queries.

query.timeout

This setting is available in Security Analytics 10.5 and later versions.

Query levels have been replaced in Security Analytics 10.5 and later with per user account query timeouts. For trusted connections, these timeouts are configured on the Security Analytics server. For accounts on Core services, there is a new config node under each account called **query.timeout**, which is the maximum amount of time in minutes that each query can run. Setting this value to zero means no query timeout will be enforced by the Core service.

max.where.clause.sessions

This setting is available in Security Analytics 10.5 and later versions.

This setting imposes a limit on how many sessions can be scanned by a single query. For example, if a user selects all meta from the database, the database stops processing results once the number of sessions read for the query reaches this configuration value. The value of 0 disables this limit.

The number of sessions needed to fully process a query is equal to the number of sessions that match the WHERE clause of the query, assuming that all terms in the where clause have a suitable index. If there are terms in the where clause that are not indexed, the database has to read more sessions and meta, and reaches this limit sooner.

max.query.groups

This setting is available in Security Analytics 10.5 and later versions.

This setting imposes a limit on the number of unique groups collected in a single query. For example, if a query has a group by clause with multiple metas that have high unique value counts, the amount of memory needed for that query could easily outpace the amount of RAM available on the server. Thus, this limit exists to prevent out-of-memory conditions from happening.

Setting a value of 0 disables this limit.

packet.read.throttle

This is a decoder-only setting that affects the access to the packets database. When packet.read.throttle is set to a value greater than 0, the decoder attempts to throttle packet reads when it detects packet contention on the packet database. Higher numbers provide more throttling. Changes takes effect immediately.

cache.dir, cache.size

All Security Analytics Core services maintain a small file cache of raw content extracted from the device. These parameters control the location (cache.dir) and size (cache.size) of this cache.

parallel.values

This setting is available in Security Analytics 10.5 and later versions.

This setting allows SDK-values operations to be executed in parallel. If this is set to 0, it will disable parallel execution. If it is set to a value greater than 0, it represents the number of threads created when each SDK-values operation is executed. The maximum value is the number of logical CPUs available when the process started.

Setting a higher value for parallel.values is useful when there are small numbers of simultaneous users, since it will allow for more complex Investigations to be executed more quickly. If there are many simultaneous users, it is better to use a low value here, since there will be many independent SDK-values operations executed simultaneously.

parallel.query

This setting is available in Security Analytics 10.5 and later versions.

This configuration is similar to the parallel.values setting in that the maximum value is the number of logical CPUs. Setting parallel.query to a specific value should take into account the number of simultaneous users to maximize CPU utilization without consistently exceeding available resources.

Setting a higher value for parallel.query is useful when there are small numbers of simultaneous users and queries, since it will allow more complex queries to be executed more quickly. If there are many simultaneous users and queries, it is better to use a low value, since there will be many independent SDK-query operations executed simultaneously.

Query operations are limited by the meta database read rate, so setting parallel.query to a value higher than 4 is unlikely to produce dramatically better results than the default value of 0. The best number to use for parallel.query will depend on the type of storage attached. Experiment with different values of parallel.query to determine the best results for your storage system.

Per-User Configuration Nodes

This topic describes the per-user configuration nodes. 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. For more information on Where Clauses, see Queries.

query.level

This setting is available in Security Analytics 10.4 and earlier versions.

The query.level setting assigns the query level that the users 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.

query.timeout

This setting is available in Security Analytics 10.5 and later versions.

The query.timeout setting assigns the maximum amount of time in minutes that a user can run each query. For trusted connections, these timeouts are configured on the Security Analytics server. For accounts on Core services, this setting is stored in the configuration tree at **/users/accounts/<username>/config**, where **<username>** is the name of the user to which the setting applies. When this value is set to zero, the Core service does not enforce the query timeout.

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 guide.

Scheduler

This topic provides a brief introduction to the scheduler and explains how to schedule commands. All Security Analytics Core services come with a built-in scheduler found under **/sys/config/scheduler**. To use the scheduler, you add the command you want to run periodically using one of two messages:

`/sys/config/scheduler addInter` - Add a command to run at the specified interval (every N hours, minutes or seconds)

or

`/sys/config/scheduler addMil` - Add a command to run at the specified time of day or even specific days of the week

Example

For example, suppose that you have a use case to delete all packet data that is greater than seven days old. Since you cannot configure the packet.dir setting to rollout data based on a time interval, you need to schedule the **/database timeRoll** command to run every so often. For this example, create a timeRoll to run every 20 minutes:

```
addIter minutes=20 pathname=/database msg=timeRoll params="type=packet
days=7"
```

This command adds a scheduled task (it is persisted between restarts of the service) to run every 20 minutes, on the **/database** node, and ages out all packet data older than seven days. The `params` parameter is used to pass all the parameters to the command specified (in this case `timeRoll`). Notice how it quotes all the embedded parameters (`type` and `days`) so they are not interpreted as parameters to be passed to the outer `addIter` command. If the parameters inside `params` need to use quotes, you must escape the inner quotes with a backslash. You can rewrite it with embedded quotes, which does not alter the command in any way:

```
addIter minutes="20" pathname="/database" msg="timeRoll" param-
s="type=\"packet\" days=\"7\""
```

This command works identically to the original, but demonstrates how to escape complicated parameter passing. Additional useful scheduler commands are:

`/sys/config/scheduler print` - Print all scheduled commands (you can also see them by doing an `ls` on the scheduler node).

`/sys/config/scheduler delSched` - Delete a scheduled command by passing in the identifier shown in the `print` (or `ls`) command.

This is a brief introduction to the scheduler. For more information on command parameters, send the `help` message to the scheduler node and pass in the command name via the `msg` parameter. For more information, see the **Services Explore View** topic in the *Host and Services Getting Started Guide*.

Rollover

This topic describes the two rollover mechanisms. 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 is 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 built-in scheduler of the Core service. The user can also explicitly request it.

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 does 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.

The **sizeRoll** command has the following parameter syntax:

```
size-roll-params    = {type-param, space}, (max-size-param | min-free-
param | max-percent-param), {max-size-warm-param, space}
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 ;
max-size-warm-param = "maxSizeWarm=", 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` parameter sets a current maximum size of the database or index. If the database is larger than this size, oldest data is deleted first (or moved to the *Warm* or *Cold* tier, depending on the configuration) 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 `maxSizeWarm` parameter sets a current maximum size on the *Warm* tier, but otherwise behaves identically to the `maxSize` parameter. When data is rolled out on the *Warm* tier, it is moved to the *Cold* tier (if configured) or deleted.

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 current size utilization of the database will have met the criteria specified by the `maxSize`, `maxPercent`, or `minFree` parameters and the optional `maxSizeWarm`. Therefore, this operation can be scheduled periodically to ensure that the database can continue to operate uninterrupted.

Example

The following example shows a typical sizeRoll scheduler entry for an Archiver:

```
pathname=/index minutes=5 msg=sizeRoll params="type=meta,session,packet
maxSize=25TB maxSizeWarm=150TB"
```

This scheduler entry specifies that every five minutes the database ensures that the max size of the meta, session, packet, and index does not exceed 25 terabytes on the *Hot* tier and does not exceed 150 terabytes on the *Warm* tier.

Queries

This topic covers the database query syntax. There are three main mechanisms for performing queries in the database, the `query`, `values`, and `msearch` calls on the **/sdk** folder on each Core service.

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 an aggregate function such as count.

The msearch call takes text search terms as it's input, and returns matching sessions that match the search terms. It can search within indexes, meta, raw packets, or raw logs.

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 0 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 returning a large number of results from the database. Their usage mostly benefits developers who are writing applications directly against the Security Analytics Core database. Normally, results are returned in the order of oldest to newest data (higher meta IDs are always more recent). In order to return results from most recent to oldest, reverse the IDs such that `id1` is larger than `id2`. This has a slight performance penalty, because the where clause must be completely evaluated before processing in reverse order can begin.

When size is left off or set to zero, the system streams back all results without paging. For the RESTful interface, this results in the full response to be returned with chunked-encoding. The native protocol returns the results over multiple messages.

The `query` parameter is a query command string with its own Security Analytics specific syntax:

```
query-string      = select-clause {, where-clause} {, group-by-clause {,
order-by-clause } } ;
select-clause     = "select ",  ( "*" | meta-or-aggregate {, meta-or-
aggregate} ) ;
where-clause      = " where ",  { where-criteria } ;
meta-or-aggregate = meta_key | aggregate_func, "(", meta_key, ")" ;
aggregate_func    = "sum" | "count" | "min" | "max" | "avg" | "distinct"
| "first" | "last" | "len" | "countdistinct" ;
group-by-clause   = " group by ", meta-key-list
meta-key-list     = meta_key {, meta-key-list}
order-by-clause   = " order by ", order-by-column
order-by-column   = meta-or-aggregate { "asc" | "desc" } {, order-by-
column}
```

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
countdistinct	Returns the number of unique values seen. Countdistinct is equivalent to the number of metas that would have been returned by the distinct function.
first	Returns the first value seen

Function	Result
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 example, the word "NetWitness" returns a length of 10. All IPv4 fields, like ip.src, 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-criteria    = 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

The following table describes the function of each operator.

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 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.

Operator	Function
length	Matches sessions that contain text meta values of a certain length. The expression on the right side must be a non-negative number.

Text Values

The system expects quoted text values. Although unquoted strings may work, the values expressed may be ambiguously interpreted as numbers or dates.

It is also important to quote any text value that 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 Security Analytics Core, dates are represented using Unix epoch time, which is the number of seconds since Jan 1, 1970 UTC. In queries, you can express the time as this 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`. A three-letter abbreviation represents the month. You can also express the Month as a two-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 already entered the index.

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 that hyphens within range expressions are not within quoted text.

Group By Clause (since 10.5)

The query API has the ability to generate aggregate groups from the results of a query call. This is done using a GROUP BY clause on the query. When GROUP BY is specified, the result set for the query is subdivided into groups. Each group of results is uniquely identified by the meta values indicated in the group by clause.

For example, consider the query `select count(ip.dst)`. This query returns a count of all ip.dst metas in the database. However, if you add a group by clause, like this: `select count (ip.dst) group by ip.src`, the query returns a count of the ip.dst metas found for each unique ip.src.

As of Security Analytics version 10.5, you can utilize up to 6 meta fields in a group by clause.

The group by clause shares some of the same functionality as the values call, but it offers significantly more advanced groups at the expense of longer query times. Producing the results of a grouped query involves reading the meta from the meta database for all sessions that match the WHERE clause, while a values call can produce its aggregates by reading the index only.

The contents of each group returned by the query are defined by the select clause. The select clause can contain any of the aggregate functions or meta fields selected. If multiple aggregates are selected, the result of the aggregate function is defined for each group. If non-aggregate fields are selected, the meta fields are returned in batches for each group.

The result set of a group by query is encoded with the following rules:

1. All metas associated with a group are delivered with the same group number.
2. The first metas returned to the group identify the group key. For example, if the group by clause specifies group by ip.src, then the first meta of each group will be an ip.src.
3. The normal, non aggregate metas are returned after the group key, but they all will have the same group number as the group key metas.
4. The aggregate result meta fields for each group are returned next.
5. All fields within a group are returned together. Different group results will not be interleaved.

If one of the GROUP BY metas is missing from one of the sessions matched by the where clause, that meta field is treated as a NULL for the purposes of that group. When the results for that group are returned, the NULL-valued parts of the group key will be omitted from the group's results, since the database has no concept of NULL.

The semantics of a GROUP BY query differ from a SQL-like database in terms of what meta fields are returned. SQL databases require you to explicitly select the group by columns in the select clause if you want them to be returned in the result set. The Security Analytics Core database always implicitly returns the group columns first.

A query with a GROUP BY clause honors the result set size parameter, if one is provided. However, due to the nature of the grouping, it puts an additional burden on the caller to page and reform groups if a fixed-size result set is requested. For this reason, you should not specify an explicit result size when making a group by call. By not specifying an explicit size, the entire result set will be delivered as partial results.

The following table describes the database honors configuration parameters that limit I/O or memory impact of a group by query.

Parameter	Function
/sdk/config/max.query.groups	This is the limit on how many groups can be held in memory to calculate aggregates. This parameter allows you to limit the overall memory usage of the query.
/sdk/config/max.where.clause.sessions	This is the limit on how many sessions from the where clause can be processed in a query. This parameter allows you to set a limit on the number of sessions that have to be read from the meta and session databases to resolve a query.

Order By Clause (since 10.5)

An order by clause can be added to a query than contains a group by clause. The order by clause causes the set of grouped results to be returned in sorted order.

An order by consists of a set of items to sort by, in ascending or descending order. Sorting can be performed on any data field that will be returned in the result set. This includes meta specified by the select clause, aggregate function results specified by the select clause, or group by meta fields.

The order by clause can sort over many columns. There is no limit on the number of order-by columns allowed in the query, but a practical limit exists in that each of the order-by columns must refer to something returned by the select clause or group by clause. The multiple column sort is imposed lexicographically, meaning that if two groups have equal values for the first column, then they are sorted by the second columns. If they are equal in the second column, they are sorted by the third column, and so on for however many order by columns are provided.

The Security Analytics Core database is unique in that the groups of results returned by a query may each have many values for a selection. For example, it is possible to select all meta that matches a meta type and organize it into groups, and it is possible to use the distinct() function to return groups of distinct meta values. If an order by clause references one of the fields in the group that multiple values, the sorting order is applied as follows:

1. Within each group, the fields with multiple matching values are ordered by the ordering clause
2. All the groups are sorted by comparing the first occurrence of the ordered field found within each group

The order by clause is only available in queries that have a group by clause, since groups are required to organize the meta fields into distinct records. If you wish to sort an arbitrary query as if there were no grouping applied, use group by sessionid. This ensures that results are returned in groups of distinct sessions or events.

Group by clauses are naturally returned in ascending group key order, but an order by clause can be used to return groups in a different order.

If an order by columns does not specify `asc` or `desc`, the default ordering is ascending.

Examples:

```
select countdistinct(ip.dst) GROUP BY ip.src ORDER BY countdistinct
(ip.dst)
select countdistinct(ip.dst) GROUP BY ip.src ORDER BY countdistinct
(ip.dst) desc
select countdistinct(ip.dst),sum(size) GROUP BY ip.src ORDER BY sum
(size) desc, countdistinct(ip.dst)
select sum(size) GROUP BY ip.src, ip.dst ORDER BY ip.dst desc
select user.dst,time GROUP BY sessionid ORDER BY user.dst
select * GROUP BY sessionid ORDER BY time
```

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}, {space, aggregate-func-param},
{space, aggregate-field-param}, {space, min-param}, {space, max-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 ? ;
aggregate-func-param  = "aggregateFunction=", { aggregate-func-flag } ;
aggregate-func-flag   = "count" | "sum" ;
aggregate-field-param = "aggregateFieldName=", meta-key ;
min-param             = "min=", meta-value ;
max-param             = "max=", meta-value ;
```

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

The following table describes the function of each parameter.

Parameter	Function
fieldName	This is the meta key name for which you retrieve unique values. For example, if fieldName is ip.src, this function returns the unique source IP values in the collection.
where	This is a where clause which filters 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 are returned. For information on the where clause syntax, see *Where Clauses*.

Parameter	Function
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 described in the following Values Flags section.
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.
aggregateFunction	Optional parameter used to change the default behavior from counting sessions, packets, or size to counting or summing the numeric field defined by aggregateFieldName. Both parameters must be specified when either is defined. Pass either `sum` or `count` to specify which behavior to perform.

Parameter	Function
aggregateFieldName	The meta field to perform the aggregateFunction on. Both aggregateFunction and aggregateFieldName parameters must be specified when the aggregate flag is set. Performing a values call using one of the aggregate functions can be significantly slower than a values call that collects totals of sessions, packets, or size. The reason for this is that each session that matches the where clause must be retrieved from the meta database. This scan causes a large portion of the query to be I/O bound on the meta DB volumes. The time taken to run an aggregate values call is linearly proportional to the number of sessions that match the where clause.
min, max	The minimum and maximum value that should be returned from the call. These parameters are used to iterate (or page) over an extremely large number of values, typically more values than could be returned from a single call. Primarily used in conjuction with the flags `sort-value,sort-ascending` such that the highest value returned would be used in a subsequent call as the `min` parameter value. The values are exclusive. If min="rsa" was specified and rsa was a valid value, rsa would not be returned, but the next highest value would be returned.

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 as shown in the following table.

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 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 Security Analytics. 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 requests unique values for ip.src. It requests unique values of ip.src in the time range given. It asks for the count of sessions that match each ip.src, and the results are the top 20 ip.src values when sorted by the number total count of sessions in descending order. In addition, the Navigation view has a meta ID range in order to provide an optimization hint to the query engine.

Msearch call

The index provides a low-level `msearch` function to perform text searches against all meta types. This type of search does not require users to define their queries in terms of known meta types. Instead, it searches all parts of the database for matches. Msearch is used by the Events view text search. See the **Filter and Search Results in the Events View** topic in the *Investigation and Malware Analysis Guide* for detail on the accepted search forms and examples.

`msearch` parameters:

```
msearch-params  = search-param, {space, where-param}, {space, limit-
param}, {space, flags-param};
search-param    = "search=", ? free-form search string ? ;
where-param     = "where=", ? optional where clause ? ;
limit-param     = "limit=", ? optional session scan limit ? ;
flags           = "flags=", {msearch-flag, {"," msearch-flag} };
msearch-flag    = "sp" | "sm" | "si" | "ci" | "regex" ;
```

The msearch algorithm works as follows:

1. A set of sessions is identified from the index by finding the intersection of three sets:
   - (Set 1) All sessions in the database
   - (Set 2) Sessions that match the where clause parameter
   - (Set 3) If the `si` flag is specified, sessions that indexed values that match the search string parameter.
2. If the search specifies the `sm` parameter, all meta from the set of sessions identified in step 1 is read and scanned to see if it matches the search string parameter. The meta will be read from the service nearest to the point where the search was executed. For example, if the search is performed on a Broker, the meta may be read from the Concentrator nearest to the broker, but if the search is performed on an Archiver the meta will be read from the Archiver itself.
3. If the search specifies the `sp` parameter, all raw packet or log entries from the set of sessions identified in step 1 is read and scanned to see if it matches the search string parameter. The packets will be read from the service nearest to the point where the search was executed. For example, if the search is performed on a Concentrator, the packet data

will be read from the Decoder, but if the search is performed on an Archiver, the packet data will be read from the Archiver itself.

4. Matches from step 2 and step 3 are returned as they are found, up to the point where the `limit` parameter is reached. Limit specifies the maximum number of sessions for which meta and packet data will be scanned. If limit is not specified, the entire set of sessions determined in step 1 is scanned.

Msearch Flags

Flag	Description
sp	Scans raw packet data
sm	Scans all meta data
si	Does index lookups for all search parameters before scanning meta
ci	Performs a case insensitive search. Returned results are case-pre-serving.
regex	Treats the search parameter as a regular expression. Only a single regular expression can be specified, but the regular expression may be arbitrarily complex.

Msearch Index Search Mode

Using the index search mode, specified by using the `si` flag, causes results to be returned significantly faster than any other mode. The main limitation of this mode is that it only returns matches on text terms that match value-indexed meta values.

- The `si` parameter must be combined with the `sm` flag. The `si` parameter implies the search only matches indexed meta.
- The `si` parameter can be used with regex searches, however only text indexed values will match. IP addresses and numbers will not match the regex.

Msearch Tips

- Always use the where clause to specify a time range for the search.
- To search for IP address ranges, specify them in the where clause.

- Use the `limit` parameter when not using the index search mode. Without it, there will be an extremely large amount of data read by the meta and packet databases.

Stored Procedures

The query and values calls provide more low-level search functionality. For more advanced use cases, server-side stored procedures exist.

Use of Quotes in Query Syntax

The query parser does not care whether you use single or double quotes within a query statement. A single- or double-quoted value is treated as text meta.

The query parser attempts to make sense of whatever you put in the statement. It is not very strict about what it will accept.

For example:

`reference.id=4752`

This clause identifies sessions that have a reference.id meta value that has a _numeric_ value of 4752.

`reference.id='4752'` or `reference.id="4752"`

This clause identifies sessions that have a reference.id meta value that has a _string_ value of "4752".

However, the query engine implicitly compares numbers and strings that look like numbers as equal when the values are semantically the same. So it works with either syntax.

For most efficient performance, however, it is always a good idea to construct the queries such that the query syntax matches the data types generated by the parser.

For example, if the parser is creating reference.id as a numeric data type (such as uint32 or uint64), then use the numeric syntax.

If the parser is creating reference.id as a text data type, then use the string syntax.

Index Customization

This topic describes how to use the custom index file to customize the index. Each Security Analytics 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 use 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 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.

You can make changes to the custom index file 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: 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 an 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" form-
at="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.

The next few sections 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. Therefore, 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 generates meta items, it assigns a data type. 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. Each parser must agree on the data format for each meta key generated by the Decoder.

When adding a custom index to the Concentrator, the data type of the custom index must match the format of the data generated by the Decoder. If the types do not match, the Concentrator attempts 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 work together as part of a Security Analytics installation, they must all agree on the types for each meta key. Conflicts of meta types between Security Analytics Core services can lead to undefined behavior.

The following table shows the metadata types supported by the Security Analytics Core services.

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

Type	Size in bytes	Description
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 takes 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 the 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 indicates that the index only keeps track of sessions that contain meta items with this meta key name. However, it does 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 parses packets 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. Therefore, anyone can create unique meta values 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. As the number of unique values increases, the rate at which new meta is indexed can decrease, and the speed with which queries are completed decreases. Since any person can influence the creation of 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. Therefore, a malicious user cannot flood the system with a large number of unique values in an attempt to make the Security Analytics system not work.

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

The value max 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 Security Analytics Core service. As of 10.3, the recommended ceiling for value max is 5,000,000 for any meta key. If there are a lot of custom indices, then the value max may have to be lower.

maxLength

The max length parameter is used exclusively on the `word` meta type. It must match the corresponding setting for **/decoder/parsers/config/token.max.length** on the Log Decoder service that is generating word token metas. The index uses the maxLength to properly interpret search terms fed into the `msearch` SDK function.

Optimization Techniques

This topic describes optimization techniques for the Security Analytics Core database. The Security Analytics Core database is set up to work with a wide variety of work loads by default. However, like any database technology, its performance can be very sensitive to both the nature of the data being ingested, and the nature of the searches that the user performs against the database.

Thresholds

Thresholds are a useful optimization that can have a dramatic effect on how fast results are returned to the Security Analytics Navigation tool. Thresholds are applied to the values call. For more information about the values call, see Queries.

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 match the where clause is very large. For example, selecting all the log events for just one hour running at 30,000 events per second matches 108,000,000 sessions.

RSA introduced the threshold feature 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 very 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.

Complex Where Clauses

The amount of time it takes for the Security Analytics 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 following table, 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 matches a large range of values
begins, ends, contains	Linear in terms of the number of unique values for 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 affect on 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.

As a general rule of thumb, the more AND clauses in the query, the faster it completes, but the more OR clauses in the query, the slower it completes.

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 users are 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 arises 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 creates meta in the meta key network with the value internal for any IP address in the 10.0.0.0/8 network.

The where clause could be expressed as:

```
network = "internal"
```

Assuming there is a value-level index on the network meta, the index does not have to expand this query into anything more complex, and the sessions matching the desired subnet are 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 users are attempting to find the host name within a URL somewhere in the session. The users might write a where clause such as:

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

In this scenario, it is likely that the url meta key contains one unique value for every session that was captured by the Decoder, and therefore has a huge number of unique values. In this case, the contains operation is 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 the URL into its 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`. Queries could be performed using:

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

Since the substring operator is no longer needed, the query is 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:

- Each save point represents a place where the index could be recovered in the case of a power failure.
- Periodically saving 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, for Security Analytics 10.5 and later installations, a save is performed on the Core index every time 600,000,000 sessions are added to the database. This interval is set by the index configuration parameter `save.session.count`. For more information, see Index Configuration Nodes.

Older versions of Security Analytics Core, or systems that have been upgraded from Security Analytics versions prior to 10.5, use a time-based save schedule that saves the index every 8 hours. You can see the current save interval by using the scheduler editor in the Security Analytics Administration UI for the service. The default entry looks like this:

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

By adjusting the interval, you can 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 exist. This has a positive effect on query performance, because it becomes less likely that queries 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 are created.

Working with Value Max

The value max limitation can be frustrating to customers who want to index all possible unique meta. Unfortunately that is not possible in the general case. Meta keys exist 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 using 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 uses value level indices in the where clause where possible, but meta database scanning is used to resolve unique values for the values call. This approach works 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 users can perform a database scan on their queries to ensure no relevant values were dropped. This feature is accessible in the Investigator 9.8 client via the right-click menu on the Navigation view. Although the meta database scan takes a long time, it reassures 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 Security Analytics Navigator view 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 Security Analytics Core service.

Index Rebuild

In rare cases, a Core service might benefit from an index rebuild. Examples:

- The Security Analytics Core service has index slices created by a very old version of the product and has not rolled out any data in more than six months.
- The index was configured incorrectly, and the customer wants to re-index all meta with a new index configuration.
- The traffic load into the Core service was very light, and the save interval was large, causing more slices than needed to be generated.

In these cases, an 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 takes 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 Security Analytics Core database. Retention refers to the period of time that 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 the age of the oldest file in the database. 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 Security Analytics Core service is not possible, then it may be necessary to use the compression options on the packet and meta database to reduce the amount of data each database writes.

If meta retention is a concern, you may want 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 custom config, or perhaps override 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 using group aggregation. Group aggregation allows a single Decoder to feed sessions to multiple Concentrators or Archivers in a load-balanced manner. Group aggregation enables the query and aggregation workload to be split among an arbitrarily large pool of hardware.

For more information, see the **Group Aggregation** topic in the *Deployment Guide*.

Grouping Workloads

The Security Analytics 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.

It is not always possible for all users to be working on the same time period simultaneously. The Security Analytics Core database can handle that use case, but it is 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 percent of the database and less than 10 percent of the index fits in RAM.

To make Security Analytics work for the customer, it is important to get the customer to organize their users into groups that 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 is another user group that does queries further back in time as part of an investigation. And perhaps another set of users do 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 and expensive concentrator intended to meet all needs.

Cache Window

Consider this sequence of events:

1. At 9:00 a.m., user "kevin" logs in to a Concentrator and requests a report on the last one 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 in to the same Concentrator and also requests a report on the last one 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 reports for Kevin could not be re-sent to Scott, since the time ranges are slightly different. 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. Re-evaluating the above action changes the sequence of events.

1. At 9:00 a.m., user "kevin" logs in to a Concentrator and requests a report on the last one 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 in to the same Concentrator and also requests a report on the last one 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 one 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 in step 3 falls in the cache window, so it is returned instantaneously. This gives 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 Security Analytics Core database for a very long time, the Core service dedicates 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 using more than their share of the Core service resources, it is a good idea to put time limits on the queries run by normal users.

Appendix A: Statistics

This topic describes statistics used to monitor system operation. The Core services provide 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.

Statistics in /database/stats

The following table shows the meaning of the statistics in **/database/stats**.

Statistic	Meaning
meta.bytes, packet.bytes, session.bytes	The total size of data (in bytes) 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 (in bytes) for all directories used by each database
meta.free.space, packet.free.space, session.free.space	The approximate total unused space (in bytes) across all directories used by each database

Statistics in /index/stats

The following table shows the meaning of the 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
index.last.load.time	The timestamp when the current index configuration was loaded from the index configuration 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

Statistic	Meaning
time.begin	The oldest time meta indexed
time.end	The most recent time meta indexed

Statistics in /sdk/stats

The following table shows the meaning of the statistics in **/sdk/stats**.

Statistic	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
values.calls	The number of calls made to the "values" function since the process was started
values.calls.cached	The number of calls made to the "values" function that were resolved by the values call result cache

Per-query statistics

SDK operations, such as 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.

The following table shows the meaning of **per-query statistics**.

Statistic	Meaning
channel.path	This stat provides a link to the connection channel over which the operation is communicating. This channel is used to communicate 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 execution is currently occurring
running.since	The time at which the query began execution
user	The user name that executed the query

Appendix B: Index Inspect

The Security Analytics Core database index has a built-in debugging feature called **inspect** that provides detailed information about the composition of its indexes. The index inspect feature is located at **/index/inspect** in every Core service configuration tree. Services that do not actually have an index, like Broker, do not have the /index/inspect feature.

Parameters

Options

Type: String
This parameter may be set to the value `all-slices` to collect inspect information about every slice in the index. If it is not set, information on the current, most recently created slice is returned.

Caution: Collecting information on all slices may take a very long time to complete if there are many index slices.

Response

Inspect returns many rows of key value pairs that represent the state of the index.

Slice Summary

The first row returned for every slice is a summary with the following values.

session1	The first session ID indexed in the slice
session2	The last session ID indexed in the slice
meta1	The first meta ID in the first session indexed in the slice
meta2	The last meta ID in the last session indexed in the slice

Per-Index Summary

There will be per-index summary rows returned for each index. Only value-level indexes are reported.

key	The meta key name for the index

pathname	The path on disk to this index
values	The number of unique values stored in this index
summaries	The number of summary entries occupied by this index in the summary.db file
pages	The number of page entries occupied by this index in the page.db file
sessions	The number of sessions that had a value that was inserted into this index
size	The cumulative "size" meta values for all sessions that inserted a value into this index
packets	The cumulative count of packets for all sessions that inserted a value into this index
summary1	The first summary ID used by this index
summary2	The last summary ID used by this index
session1	The first session ID referenced by this index
session2	The last session ID referenced by this index

Slice Summary Footer

The last row in each inspect report contains cumulative statistics for all the indexes in the slice.

totalKeys	The number of indexed meta types
totalValues	The number of unique values tracked by all indices in this slice
totalMemory	An approximate total of the memory needed to open this index slice