

A COMPUTER-IMPLEMENTED METHOD, AND A SERVER

Technical Field

The invention relates generally to the field of data storage. One aspect of the invention relates to a computer-implemented method. Another aspect of the invention relates to a server.

Background

In terms of data storage regimes, it may be useful for data, such as data relating to prior transactions, to be stored in a database. An example of such a database is noSQL.

For example, US8433695 discloses a database with a key/value store comprising a first tier storage device configured to store information about a plurality of keys for a plurality of values without the values, and a second tier storage device coupled to the first tier storage device and configured to store the values associated with the keys without the keys, wherein the first tier storage device has lower latency and higher throughput than the second tier storage device, and wherein the second tier storage device has higher capacity than the first tier storage device.

US11100123 discloses identifying keys having common values. As an example, one key-value group includes (1, [u1, u2, u3, u4]) and another key-value group includes (2, [u1, u2]). The system may identify that the values u1 and u2 are common between keys 1 and 2. Once the common values are identified, the system may generate new key-value groups having common keys and common values.

One technical problem that may exist in the art is how to optimise the placement of data based on the number of network I/O operations, non-volatile memory and/or volatile memory space. Another technical problem that may exist in the art is how to

optimize a balance between network I/O operations, non-volatile memory space and/or volatile memory space based on latency and/or cost.

Summary

Embodiments may be implemented as set out in the independent claims. Some optional features are defined in the dependent claims.

In a further embodiment there is provided a method performed in a communication server apparatus, the method comprising, under control of a processor of the communication server apparatus:

storing a key in an upper tier memory, and
storing a nested value map or a portion of a nested value map aggregated from data in a lower tier memory.

In a still further embodiment there is provided a communication server apparatus, the communication server comprising at least one processor, the communication server apparatus being configured, under control of one or more of the at least one processors, to execute instructions, to:

store a key in an upper tier memory, and
store a nested value map or a portion of a nested value map aggregated from data in a lower tier memory.

Implementation of the techniques disclosed herein may provide significant technical advantages. Advantages of one or more aspects may include:

- optimisation of database latency.
- optimisation of volatile memory and/or non-volatile memory cost.
- reduce the number of/make a trade-off on the network I/O operations an application has to do to retrieve data.

In an exemplary implementation, the functionality of the techniques disclosed herein may be implemented in software running on a server communication apparatus (such as a one or more servers, one or more virtual machines, one or more processors or a cloud computing platform), which communicates with the applications running on the user terminals, such as mobile phones. The software which implements the functionality of the techniques disclosed herein may be contained in a computer program, or computer program product. The server communication apparatus establishes secure communication channels with the user terminals for optimising which parts of the database is stored on various physical memory tiers, such as between RAM and SSD for aggregated data used by the fraud engine to assess the risk of each transaction. The techniques may also be used for any kind of analytical use case, where data aggregated over time needs to be reviewed.

Brief Description of the Drawings

The invention will now be described, by way of example only, and with reference to the accompanying drawings in which:

Figure 1 is a schematic block diagram illustrating an exemplary delivery/transportation service.

Figure 2 is a schematic block diagram illustrating an exemplary communications server for the delivery/transportation service.

Figure 3 is a flow diagram of a transaction process.

Figure 4 is a schematic diagram of hardware and software components used for the process in Figure 3.

Figure 5 is a flow diagram of an example process for structuring the database schema.

Figure 6 is a flow diagram of an example schema implementation.

Detailed Description

The techniques described herein are described primarily with reference to use in a fraud engine for a transaction platform provider. This may be useful to avoid fraud and so may minimize the risk of fraud to the system users, merchants and drivers as well. The techniques may also be used for any kind of analytical use case, where data aggregated over time needs to be reviewed.

Figure 1 shows an exemplary architecture of a system 100, with a number of users each having a communications device 104, a number of merchants each having a communications device 109, a number of drivers each having a user interface communications device 106, a server 102 (or geographically distributed servers) of a transaction platform provider and communication network 108 connecting each of the components. Each user contacts the server 102 using a user software application (app) on the communications device 104. Similarly, the drivers and merchants may use an app on their devices 106, 109.

For deliveries or e-commerce-based transactions, the user device 104 may allow the users to input queries containing the keywords for the items of interest and delivery addresses. The user may see a list of merchants and/or items provided by the merchants, and order items from the merchants. The merchant may contact server 102 using merchant device 109 for providing information about their items and receiving orders for each confirmed transaction. The driver contacts server 102 using driver device 106. The driver device 106 allows the drivers to indicate their availability to take the delivery jobs, information about their vehicle, their location. Server 102 may then match drivers to the delivery, based on, for example: geographic location of merchants and drivers, maximising revenue, user or driver feedback ratings, weather, driving conditions, traffic level / accidents, relative demand, environmental impact, and/or supply levels. The user may be offered a particular delivery cost and approximate delivery ETA. If the user accepts the offer, the system may go through a payment authorisation process. If the authorisation is

approved, the merchant will then be notified and directed to provide goods for the driver to pick up. The selected driver will then be notified and directed to the pickup location to pick up the goods. During the delivery, the user device 104, the driver's device 106, the merchant's device 109 and the server 102 may be updated with real-time trip information including the real-time location of the driver's vehicle, the destination, the driver fare and/or other trip-related information. After the trip the driver's device 106 may send a confirmation that the trip has ended to server 102. Once the transaction is approved and/or the delivery is completed, the user device 104, the driver's device 106, the merchant's device 109 and the server 102 may be updated with details of the completed financial transaction. This allows an efficient allocation of resources because the available fleet of drivers is optimised for the users' demand in each geographic zone.

For transportation, the user device 104 may allow the user to enter their pickup location, a destination address, one or more service parameters, and/or after-ride information such as a rating. The one or more service parameters may include the number of seats of the vehicle, the style of vehicle, level of environmental impact and/or what kind of transport service is desired. Each driver contacts the server 102 using a driver app on the communication device 106. The driver app allows the driver to indicate their availability to accept jobs, information about their vehicle, their location, and/or after-ride info such as a rating. The server 102 may then match users to drivers, based on, for example: geographic location of users and drivers, maximising revenue, user or driver feedback ratings, weather, driving conditions, traffic level / accidents, relative demand, environmental impact, and/or supply levels. The user may be offered a particular transport cost, or a range based on different types of vehicles, and an approximate ETA. If the user accepts the offer, the system may go through a payment authorisation process. If the authorisation is approved, the selected driver will then be notified and directed to the pickup location to pick up the user/passenger. During the trip, the user device 104, the driver's device 106, and/or the server 102 may be updated with real-time trip

information including the real-time location of the driver's vehicle, the destination, the trip fare and/or other trip-related information. At the conclusion of the trip, the driver's device 106 may send a confirmation the trip has ended to server 102. Once the transaction is approved and/or trip completed the user device 104, the driver's device 106, and/or the server 102 may be updated with details of the completed financial transaction. This allows an efficient allocation of resources because the available fleet of drivers is optimised for the users' demand in each geographic zone.

In the above examples the server 102 may be used by a transaction platform operator. Typically, the platform used by the transaction platform operator will facilitate transactions between the users, and goods or services providers such as drivers or merchants. The term "provider" may refer to a driver or merchant as described in the above examples or any other third party associated with providing a product or service for purchase, or a travel or ride or delivery service via transaction platform operator.

In the context of the present specification, "a server", "the server" or "said server" is a computer program that is running on appropriate hardware and is capable of receiving requests (e.g., from electronic devices) over a network, and carrying out those requests, or causing those requests to be carried out. The hardware may be implemented as one or more physical computers, one physical computer system, one or more virtual machines, or a cloud-based server network. In the present context, the use of the expression a "server" is not intended to mean that every task (e.g. received instructions or requests) or any particular task will have been received, carried out, or caused to be carried out, by the same server (i.e. the same software and/or hardware); it is intended to mean that any number of software elements or hardware devices may be involved in receiving/sending, carrying out or causing to be carried out any task or request, or the consequences of any task or request; and all of this software and hardware may be one server or multiple servers. The same

approach is to be taken for references to other systems, software or hardware components e.g., processor, memory, storage etc.

Here the transactions referred to are the payments done for anything the transaction platform operator offers, for e.g., Ride-hailing, food delivery, top-up, payment to merchants, payment to drivers, e-commerce etc. This can include any transaction where a user, driver or merchant has to pay to the transaction platform operator for its services or products.

Referring to Figure 2, further details of the components in the system of Figure 2 are now described. The communication apparatus 100 comprises the communication server 102, and it may include the user communication device 104, the merchant communication device 109 and the driver communication device 106. These devices are connected in the communication network 108 (for example, the Internet) through respective communication links 110, 111, 112, and 114 implementing, for example, internet communication protocols. The communication devices 104, 106 and 109 may be able to communicate through communication networks and/or protocols, including cellular communication networks, LAN, WAN, private data networks, VPN, fibre optic connections, laser communication, microwave communication, satellite communication, Bluetooth, Wifi, NFC, etc., but these are not specified in Figure 2 for the sake of clarity.

The communication server apparatus 102 may be a single server as illustrated schematically in Figure 2. Alternatively, the functionality performed by the server apparatus 102 may be distributed across multiple physically or logically separate server components. Where the term server is used, it may be understood to mean any configuration of physical or virtual processors which is configured for the desired tasks. In the example shown in Figure 2, the communication server apparatus 102 may comprise several individual components including, but not limited to, one or more microprocessors 116, one or more memories 118 (e.g., a volatile memory such

as a RAM), and/or longer-term, non-volatile, or persistent, memory 119 (e.g., SSD (Solid State Drive) or Hard disk drives (HDD)) for the loading of executable instructions 120, the executable instructions defining the functionality the server apparatus 102 carries out under the control of the microprocessor 116. The communication server apparatus 102 also comprises one or more input/output modules 122 allowing the server to communicate over the communication network 108. One or more user interfaces 124 are provided for administrator control and may comprise, for example, computing peripheral devices such as display monitors, computer keyboards and the like.

The server apparatus 102 may also comprise one or more databases 126 stored in volatile memory 118 and/or non-volatile memory 119, for storing data, which may include data on geographic information, images, products, points of interest, users, drivers, merchants, transaction data, aggregated data or parameters, payment data, and other relevant data. The data may be stored in a data structure according to the requirements of the application, or as described in more detail below. The database 126 may be replicated, distributed, sharded or otherwise optimised according to the requirements of the application, or as described in more detail below.

The user communication device 104 may comprise several individual components including, but not limited to, one or more microprocessors 128, a memory 130 (e.g., a volatile memory such as RAM), and/or longer-term memory such as flash memory or SSD (Solid State drives) for the loading of executable instructions 132, the executable instructions defining the functionality the user communication device 104 carries out under the control of the microprocessor 128. The user communication device 104 also comprises an input/output module 134 allowing the user communication device 104 to communicate over the communication network 108. A user interface 136 is provided for user control. If the user communication device 104 is, say, a smartphone or tablet device, the user interface 136 will have a touch panel display as is prevalent in many smartphones and other handheld

devices. Alternatively, if the user communication device 104 is, say, a desktop or laptop computer, the user interface 136 may have, for example, computing peripheral devices such as display monitors, computer keyboards and the like.

The merchant communication device 109 may be, for example, a smartphone or tablet device with the same or a similar hardware architecture to that of the user communication device 104.

The driver communication device 106 may be, for example, a smartphone or tablet device with the same or a similar hardware architecture to that of the user communication device 104. Alternatively, the functionality may be integrated into a bespoke device such as a taxi fleet management terminal.

Thus, it will be appreciated that Figures 1 to 4 and the foregoing description illustrate and describe a system comprising:

- a communication server 102;

- at least one user communication device 104 having an associated user and configured to initiate a transaction which includes transaction data; and
- communication network equipment 108 configured to establish communication with the communications server 102, and the at least one user communication device 104;

- wherein the communications server 102 comprises a at least one processor, an upper tier memory, and a lower tier memory, the server being configured, under control of one or more of the at least one processor 116, to execute instructions stored in the upper-tier memory 118 or the lower-tier memory 119 to:

- store a key in the upper tier memory 118, wherein the key is derived from one or more entity identifiers in the transaction data;

- store a nested value map or a portion of a nested value map aggregated from the transaction data in the lower tier memory 119, and

determine whether to approve or deny the transaction based on the nested value maps;
wherein the user communication device is further configured to indicate to the user whether the transaction has been approved or denied.

The one or more entity identifiers used to derive or construct the key can be selected from the group consisting of user id, IP address, device id, merchant id, and any combination thereof. By way of non-limiting example, the key may be just the user id, or may be a combination of user id and device id, as will be explained in further detail below. Where the key is derived from a combination of entity identifiers, the identifiers may be combined in any suitable fashion, such as by concatenation (with or without separator symbols). In some embodiments, the key may be a transformed version of the entity identifier or combination of entity identifiers, such as a hash.

Further, it will be appreciated that Figure 6 illustrates and describes a method performed in a communication server apparatus 102, the method comprising, under control of a microprocessor 116 of the communication server apparatus 102:

storing a user key associated with a user in an upper tier memory 118,
and
storing a nested value map or a portion of a nested value map aggregated from a transaction data in a lower tier memory 119.

Fraud Engine

In general, it may be useful for the transaction platform operator to use a Fraud Engine to maintain data points related to transactions in the system. Data analysts then use these data points to write rules for different use cases to prevent financial fraud. Some examples of these data points could be, the number of devices from which the user has done a successful transaction, the number of users using the same card to transact, etc. To detect fraud, the analysts feed these data points into

predefined rules which give out a verdict of whether a transaction should be allowed or denied. For this, the Fraud Engine must have data to determine in each transaction the risk of fraud. The data points can be determined in real-time from a database of transactions, the data points may be directly requested from pre-computed aggregated data, a machine learning model may be used to give an allow/deny verdict, or other methods according to the requirements of the application.

Also, merchants, drivers and/or users may benefit from the transaction platform operator's use of a Fraud Engine, through reduced fraud, lower costs and greater trust.

In this example, the Fraud Engine is used by the transaction platform operator, but it could also be used by the Payment Service Provider (PSP), payment networks, or other parties to the transaction. The Fraud Engine operated by the transaction platform operator, is separate from and additional to any fraud risk management processes employed by other parties to the transaction. For example, typically with credit card transactions, the issuer bank may request a one-time password (OTP) via a two-factor authentication device. For example, they may send the OTP via a SMS on the user's mobile phone. Additionally, the Fraud Engine operated by the transaction platform operator may interact with the risk management processes employed by other parties. For example, if that user has previously successfully completed a 2FA OTP within a certain time period, the Fraud Engine may instruct the other parties that a 2FA OTP is not required for this transaction or may avoid doing an internally generated 2FA OTP check.

Example transaction flow

Depending on the application, different transactions may be implemented according to the requirements of that transaction. For example, Figure 3 shows an end to end flow of a top up transaction 300 done by a user to add money to an e-wallet held by a

transaction platform operator. This example is used to show how the database of transactions is generated according to one or more example embodiments, and how it can be subsequently used by the Risk Service / Fraud Engine. Each function shown in Figure 3 may be implemented by a separate server. Alternatively, one or more functions may be implemented by multiple servers, or some or all of the functions may be consolidated in a single server.

The user communication device depicted above 104 acts as the interface between user input and the information subsequently flowing through various modules on hosted on different servers. When the user enters a top-up amount for their e-wallet and clicks the button a credit/topup/card API gets invoked to the transaction platform operator's internal API Gateway 302. This call, in turn, is forwarded to a Session Manager 304 to verify the authenticity of the request, whether it is coming from a valid user or not. Here, the Session Manager 304 verifies the authenticity of a JWT token that would have been cryptographically signed and sent to the user earlier during login. The user communication device will ensure the same JWT token is passed with every server request. Once, the JWT token is verified by the internal API Gateway 302, the call is forwarded to the e-wallet wallet service 306. The e-wallet wallet service 306 then calls the Risk Service 308 to see if the transaction should be allowed/denied. If allowed, then a call is made to a Gateway Module 310 to charge the user the subsequent amount. The Gateway Module 310 maintains stats for every transaction and checks if the amount has been authorised or not. That means that it checks if the user has the required amount in his/her credit card account or not and does an authorisation hold.

If the amount is not authorised, the Gateway Module 310 initiates an auth call to a link Adapter 312. The adapter 312 then forwards the request to a Link Online 314 which is responsible for talking to Link Router 316 to determine an appropriate PSP to process the transaction (according to rules such as transaction currency, transaction type, and may further filter based on health of each PSP etc). In the current example,

the Router 316 returns Adyen 318 as a processor for top-up transactions. Once, the relevant route is determined the PSP is called with the relevant details to carry out the transaction. Consequently, the Router 316 does not talk to acquiring bank, issuer bank, card network or card scheme directly. It is done through PSPs like Adyen, Stripe, Moca, et al. Subsequently, a successful response when received from the PSP 318 is sent back to the e-wallet wallet service 306 and API Gateway 302.

After that, the API Gateway 302 resumes the original request to charge the user the relevant amount. The API Gateway 302 forwards the request to e-wallet wallet service 306 and when it eventually reaches Gateway Module 310, it is identified that the amount is on hold and not yet charged, so the Link Online 314 calls the PSP 318 to forward the request to issuing bank which is responsible for moving money from users' bank account to the transaction platform operator's bank account. Once that is successful, a successful response is propagated up to e-wallet wallet service 306 and API Gateway 302. API Gateway 302 then returns the relevant transaction success message to the user device 104.

During the whole flow, the relevant data points are collected by different services and used for their use cases. For example, when the call is initiated from the user device 104, the API gateway 302 automatically receives some basic data points like who has initiated the transaction, and for what amount and currency is the transaction initiated. These data points are then propagated to e-wallet wallet service 306, which is responsible for storing them and also proxying them to the Risk Service 308 for fraud checks. Other data points like the final status of the transaction, which can only be obtained after a transaction has been completed are sent to the Risk Service 308 via Kafka Streams asynchronously.

Risk Worker

The Risk Service / Fraud Engine 308 is shown in more detail in Figure 4. It includes a risk worker 402 which generates, updates and maintains raw data in a timescale

database 404 and generates, updates and maintains aggregated data in an aerospike database 406. A risk API 408 includes any rules and logic to determine for a new transaction what is the risk score and based on that whether to approve or decline the transaction. The risk API 408 may use the aggregated data in the aerospike database 406, or it may use raw data in a timescale database 404, to make the determination.

When the data is received by Risk Worker 402 in the various Kafka streams, for each field that needs to be stored in Aerospike 406, the corresponding path in the stream payload is stored to fetch the actual values. Once, these values are obtained, a schema can be used to interpret and store the values in Aerospike in a pre-computed format. During the whole process, the constructed key stays in volatile memory and value maps reside in non-volatile memory, as described in more detail later.

The Risk Worker 402 is responsible for consuming Kafka Stream of data points from different services and aggregating them in Aerospike 408. The Kafka Streams referred to here are generally streamed out with different data points after a transaction is completed. In the above flow for example, once the e-wallet wallet service 306 has sent a successful response to the API gateway 302 after the capture amount call, the e-wallet wallet service 306 will stream out all the data points it has collated until now on a Kafka topic. This can then be consumed by whoever has subscribed to the same topic. The data points used in the different examples of the document are consumed from various streams by the Risk Worker 402. For example, the device id is obtained from a stream owned by Patronus which is a background service running on user terminal apps and is responsible for capturing device-related data points and generating an id for it. User id, currency and status of the transaction are consumed from a stream owned by the e-wallet wallet service 306 which is responsible for collecting all the transaction-related data points sent from user terminal apps to API Gateway 302 which in turn are proxied to the e-wallet wallet service 306.

Two applications for the fraud API are to reduce chargebacks and arrears. A fraudulent user may use the ride hailing service but empties their e-wallet before the money can be charged. In this case, the user owes the transaction platform provider money which is never paid out. This is called arrears.

In another scenario, payment is made out after a user uses the service but the user then later claims to their issuer bank that it was not authorized by him/her. This is called a chargeback, where the user files a case and contests with the bank that it was a case of account takeover and the money should be refunded back.

In the event that pre-computed aggregated data is used by the fraud engine, the aggregated data must first be captured from the transaction data. This may be done in batches from time to time, e.g., every month or the aggregated data may be updated every transaction, or other methods according to the requirements of the application. For example, as shown in Figure 4 the aggregated data is updated after every transaction from the Kafka Streams. This allows us to decouple the service that needs to aggregate data from the service that needs to determine a risk score and/or verdict.

The pre-computed aggregated data may be stored in database 126 such as Aerospike (a NoSQL key-value store) 406 as part of an implementation of a fraud engine. This fraud engine may be implemented in server 102 shown in Figures 1 and 2, or more specifically as shown in Figure 4. Server 102 may store database 126 in volatile memory 118 and/or non-volatile memory 119. As mentioned above, volatile memory 118 may include different tiers such as onboard cache and/or RAM (random access memory), and/or the non-volatile memory 119 may include different tiers such as SSD (Solid State Drive) or Hard disk drives (HDD). Volatile memory 118 in general may be considered a higher tier compared to non-volatile memory 119. The ranking of tiers, is according to the relative access speed of each tier. A

higher tier has a faster access speed than a lower tier. This is described in more detail below.

Example storage of aggregated data

Based on the above, this might result in, for example, a parameter regarding how many successful transactions a user has done on a given device. In that case, that data could be stored in the following format.

"103158129.a5da5622790b32a29e3b8533cf54ccb7efc346a4.success": MAP({'c':1})

Similarly, we can have a use case in which we want to find out the amount (in USD) a user has successfully transacted on a given device. The aforementioned key-value pair can be utilized to store the same information.

"103158129.a5da5622790b32a29e3b8533cf54ccb7efc346a4.success": MAP({'c':1, "amt": 100})

In the above examples, *103158129* represents a unique identifier for a user, *a5da5622790b32a29e3b8533cf54ccb7efc346a4* represents a unique identifier for a device (mobile, laptop, tablets, et al.) and *success* represents the status of the transaction done by the user. These identifiers concatenated together by dots represent the key for our data in the key-value store. The value is stored as a map which stores the count of transactions and the overall amount that has been successfully transacted using the device mentioned in the key by a given user. Counts and amounts are represented by *c* and *amt* respectively in the value map. There can be an even larger number of entries in the value map depending on the aggregates that we want to store. Some more examples of aggregates could be *mean*, *average*, *standard deviation*, etc.

To store this data over different time windows, an Aerospike feature allows the different versions of values to be stored against a single key. These different versions are stored in buckets called bins. An example structure is:

	2022	2022-07	2022-06	2022-07-20	2022-07-19	2022-07-18
103158129.a 5da5622790b 32a29e3b853 3cf54ccb7efc3 46a4.success	MAP({'c':8, "amt": 190})	MAP({'c':5, "amt": 100})	MAP({'c':3, "amt": 90})	MAP({'c':1, "amt": 20})	MAP({'c':2, "amt": 40})	MAP({'c':2 "amt": 40})

In a transactional system, there could be millions of combinations of user ids, device ids and statuses that could be stored in the system. For example, let's assume that the current population of Southeast Asia is 682,026,125 (~682 million) and 30% of the population (~204 Million people) uses a particular merchant app for their various needs and use cases. Furthermore, each of these 204 million people has used approximately 3 devices in their life where their transaction could have been successful or failed. This would have resulted in our system storing 1.2 billion keys (204 million * 3 * 2).

According to one embodiment, the keys and the primary index created on those keys may be stored in volatile memory to access data faster. The value maps according to the current settings may be stored on non-volatile memory to optimise the cost, meanwhile achieving acceptable performance.

Primary indexes generally store all the keys and the pointer that points to the data block where a value map is stored on non-volatile memory for a given key in a data structure (for example, B-Trees, Hash Tables, et al.), making data access faster by avoiding sequential scans. In different databases, the space occupied by the primary indexes can vary according to the different data structures used to maintain them. In the current implementation with Aerospike, each key corresponds to 64 bytes of

space in the primary index so, for 1.2 billion keys this would require approximately 80 GB of space in volatile memory. As fraud checks become more sophisticated the need for pre-computing aggregates and storing them against different kinds of data points grows proportionally. With the introduction of replication for data loss prevention, this problem becomes even worse.

Generally, data stored in RAM can be accessed very quickly regardless of where in memory it is stored, so it's very fast—milliseconds fast. On the other hand, drives (including solid state storage) tend to be slower than RAM. Hard disk drives are mechanical devices, so they can't access information nearly as quickly as RAM does. Even SSDs which are the more advanced versions of traditional hard drives are much faster than hard drives but are still slower than RAM. There are primarily two reasons for that difference in speed. First, the memory chips in SSDs are about 1000 times slower than those in RAM. Second, there is a bottleneck created by the interface that connects the memory device to the computer. When it comes to cost, RAM is more expensive than drives. Generally, SSD prices can be anywhere between $\frac{1}{4}$ th to $\frac{1}{10}$ th of RAM prices and that is one of the primary reasons why everything is not stored on RAM.

To combat the ever-growing volatile memory costs for the storage of keys we devised an approach to reduce the number of keys by moving most of the data from the key (stored in more expensive volatile memory) to value maps (stored in less expensive non-volatile memory). This helped significantly in bringing down the overall memory utilization of Aerospike to store the same amount of data. For example, the same key-value pair that we discussed before,

```
"103158129.a5da5622790b32a29e3b8533cf54ccb7efc346a4.success": MAP({"c":1,  
"amt": 100})
```

would now look like as shown in Figure 5a.

Going forward we'll call this key-value pair as a record. This is stored under the set name "risk\$from.1" according to the schema discussed in the following paragraphs.

Note: In this example, all the devices and the corresponding statuses of the transaction done on those devices by user 103158129 are consolidated under one map. Prior art methods, using a longer key specific to each user, device, and status of the transaction, which would result in six separate keys for the same data.

A lot of data storage engines allow storage of key-value pairs but what is different in our approach is the architecture that we are proposing here selectively allows us to control non-volatile memory utilization and volatile memory utilization of the database and the number of network I/O operations that the application code has to do while making calls to the database. Database 126 may allow keys to be stored in volatile memory and values on non-volatile memory and supports nested maps as a data structure which can be stored as a value.

The proposed approach allows the creation of a set of records which can be represented by a schema that can be parsed to store data in the most optimal way possible, for example as shown in Figure 5b.

The first label "sets" represent an array of Aerospike Set. An aerospike Set is similar to a table in a relational database. One of the big differences is that with Aerospike, you do not need to predefine a schema. Thus, Sets are created dynamically and implicitly on the first record insertion in the Set. These Sets are used to group key-value pairs. This abstraction of the Set is important so that the overall data that we need to query can be reduced and queries can be optimised.

In the schema, in the same array represented by the label "sets", there is another label called "configs" which can be used to split the data amongst actual Aerospike Sets. For the above example, there will be four Aerospike Sets namely "risk\$from.1",

"risk\$from.2", "risk\$from.3" and "risk\$deviceId.1". Here, the naming convention that is used to define the set names is as follows, *namespaceName\$recordKey_1.recordKey_2.....recordKey_n.setNumber*.

Here *namespaceName* should be replaced by the value present for the label *"namespace"*, *recordKey_n* should be replaced by all the entities present in the array represented by the label *"recordKeyEntities"* and *setNumber* should be replaced by the numbers mentioned inside the *"configs"* label.

Inside each *"configs"* label, there can be multiple aggregates, these aggregates are stored according to the set config defined in the schema. Here, what we call aggregates is a set of entities (for example device id, user id, transaction status, et al.) against which we want to save the value that is obtained by applying aggregate functions (for example count, sum, et al.) on raw data. The numbers mentioned in the schema when combined with the entities mentioned under the *"recordKeyEntities"* label form a new set that stores the aggregates present inside that particular *"number label"*.

The entities present inside the *"an arbitrary number label"* are stored in the same nested map against a particular key (formed by joining the corresponding value of the entities stored in the *"recordKeyEntities"*). For the above example, if we consider *"set 1"* for record key entity *"from"* we can find that it has three aggregates defined, each having *"bmks"* and *"entities"* labels. The label *"entities"* is used in constructing the nested map that has to be stored for a particular record key. On the other hand, the label *"bmks"* define the leaf level keys of our nested map and represent the aggregated values. In our case, *"set 1"* defines that data would be aggregated on three combinations: all the possible values of *device id and status*, all the possible values of *device id, status and currency* and all possible values of *device id*. Each of them would store 2 key-value pairs at the leaf level to store the total count and sum of amount of all of the transactions. One such representation for a record key *"103182985"* in an

Aerospike Set *"risk\$from.1"* could be as shown in Figure 5c. The label for time windows is skipped for brevity and would and should be included in the final record.

Using the *"configs"* label avoids the need to store all of the data related to a user in an Aerospike Set called *"from"* and name it *"risk.from"*. The amount of data that would have gone into a single record for that particular Aerospike Set would have created a huge overhead of network I/O operations that had to be done just to retrieve the data from the database. Essentially, the size of the data is directly proportional to the amount of network I/O operations that have to be done to retrieve the data. For this reason, the data had to be split across multiple Aerospike Sets to optimize the retrieval times.

Figure 6 shows method 600 to arrive at the most optimal storage solution for a set of aggregates and a schema can be constructed for the same purpose. This method 600 is intended to create the schema shown previously.

To start 602 the process, all the entities and their respective expected cardinalities (for example, device ID, pax ID, status, IP, merchant ID et al.) to be aggregated are obtained as input 604.

The input is checked whether it is empty 606. If the input is empty, the process ends 608. If the input isn't empty, then all the entities are sorted 610 in in descending order according to their expected cardinalities. The sorted keys are then placed in an array called *"entityArray"*. The visual representation for the same might look like this `entityArray[n-1]..... entityArray[1], entityArray[0]`.

A check 612 is done if a configuration already exists for the set of entities or not. If it already exists, the process ends 614. We can use the same configuration for aggregation and proceed ahead.

If a configuration does not exist the base distribution of entities is defined as placing 616 all the entities under the “recordKeyEntities” label. All the elements of entityArray are placed under the “recordKeyEntities” label. But given the fact that we are making a trade-off between non-volatile memory and volatile memory, we would always want to store some entities under the “config” and “aggregates” labels.

A check 617 is done to the elements of entityArray. If the elements of entityArray are less than 2, then in the first element of the array found within the “aggregates” label of the “config” label, place 619 all the aggregate keys at the leaf level inside the “bmks” label. Additionally, place 621 all the input entities under the “recordKeyEntities” label.

If the elements of entityArray are more than 2, we’ll start with the entity having the least expected cardinality and try to move 618 it from the “recordKeyEntities” label to the “config” and “aggregates” labels. We’ll keep moving entities with the least expected cardinality from the “recordKeyEntities” label until the record size exceeds the “write_block_size” set in the Aerospike configuration. If it exceeds, then we revert the present configuration and use the last saved 640 configuration whose record size was lesser than the “write_block_size”. We also keep prepending 620 the moved elements to the “movedEntities” array.

In some embodiments, determining the record size may involve utilizing a schema to create a mock record that conforms to the specified structure. We then populate this record with dummy values representing real-world scenarios. By converting it to bytes, we can determine the worst-case record size based on the number of values present in this dummy structure.

A record set can be an Aerospike Set, a table, or a collection, depending on the specific data platform being used.

Moving the entity under “configs” and “aggregates” labels means first checking 622 if there are some aggregates present under the “configs” label or not, or in the worst case scenario, “configs” label exists or not. If the label doesn’t exist, then we create 624 the “configs” label and assign it an auto-incrementing number as a unique identifier. This number is later used to identify sets uniquely having the same entities in the “recordKeyEntities” label. An “aggregates” label is placed 626 inside “configs” label which is an array of aggregates. Inside the “aggregates” label, the first element is initialized with a label called “entities” with the entities present inside the “movedEntities” array. And the aggregate operation that has to be performed is stored inside the “bmks” label.

Each Aerospike Set config is iterated over 628 and the corresponding aggregates stored under the “aggregates” label to check 630 if the entities present in “movedEntities” overlaps with the entities present under the “entities” array. Here overlapping means that when both the arrays are parsed from start to end, some elements are present in the same indexes in both the arrays. In mathematical terms, elements present up to i^{th} index in both the arrays should be same that is $\text{movedEntities}[0 \dots n-i]$ is exactly same as $\text{entities}[0 \dots n-i]$ where both the arrays should have at least one element. If that’s the case, then we create 632 another element under the same “aggregates” label and place the corresponding aggregate operations in “bmks” label and the entities in the “entities” label. It needs to be checked 633 if the expected record size for that Aerospike Set configuration is less than the declared “write_block_size” in Aerospike configuration. If not, we remove 634 the previously created element under the aggregates array and then create 624 another config entry under the “configs” label and store 626 the entities (under “entities” label) and leaf level keys (under “bmks” label) in the first element of array represented by “aggregates” label.

There could be an edge case if we have n entities and all of them get moved inside to config and aggregates because the record size is less than `write_block_size`. In that case, we would want to keep the one with the most cardinality under the "recordKeyEntities" label and the rest of them in the "entities" label inside the "aggregates" and "config" labels. So, there is always one key against which we can store a value map.

If it is checked 635 that the record size is more than the "write_block_size", then we end 638 the process and use 636 the last saved configuration. This signifies that no more keys can be moved inside the "configs" and "aggregates" labels.

For example, if we get an input to aggregate based on the `user_id`, `device_id`, currency and status, we would go through the defined process and sort the keys in descending order according to their expected cardinalities. The expected order of entities post sorting would look something like this, `user_id`, `device_id`, currency and status. Subsequently, we'll follow the algorithm to move all the keys from "recordKeyEntities" label to "configs" label. The algorithm would be able to move `device_id`, currency and status to the "aggregates" array under "configs" label. If we get inputs to then create aggregates for `user_id` and `device_id` then same process would be followed and `device_id` would be placed under the same "aggregates" array because `device_id` is overlapping with the previously placed aggregate `device_id`, currency and status. This particular part of the algorithm is important because it ensures we can make use of the already existing records to store new records and thus optimise and make best use of drive memory.

But in another scenario, if the algorithm gets the request to create an aggregate on `user_id`, `device_id` and status then the algorithm would have created an "aggregates" array with `device_id` and status. But then subsequently if another input comes for `user_id`, currency and status, the algorithm would not be able to place it in the same "aggregates" array because when both the arrays are sorted in descending

order based on cardinality, there is no overlapping element in when the arrays are parsed from start to end.

The aforementioned scenario can be best described with this example, where if we want aggregates for all the possible values of device id and status, all the possible values of device id, status and currency and all possible values of device id the record can be stored like shown in Figure 5d, instead of having three different records like shown in Figure 5e.

If then we get a new input of entities which require us to create an aggregate for user_id, ip and status, we would again repeat the described methodology. But since the cardinality of ip can be huge and the overall record size can exceed aerospike write_block_size this aggregate would be placed in a new "entities" label under the "configs" and "aggregates" labels.

There could be another edge case where we want to aggregate on user merchant pair and the cardinality for the merchant with whom a user has interacted in a year can be quite high. In that case none of the flows would be able to cater to the use case and place merchants inside the "config" and "aggregates" labels. In that case, there would be no last configuration to fall back to. We would want to keep both merchant ids and user ids in the "recordKeyEntities" label and the aggregate keys in the "bmk" label. In such cases "entities" label might/might not remain empty.

In another scenario, there might not be two entities to aggregate. The number of entities obtained as input can be less than 2. In this case, we want to directly place the entity inside the "recordKeyEntities" label and the rest of the aggregate keys in the "bmk" label of the first "aggregates" label that is found corresponding to the same "recordKeyEntities" label. In this case, we don't want to check size, because we are just storing aggregate labels in the value map which won't contribute much to the record size.

Evaluating to find out what the worst-case cardinality for an *entity* may be implemented as described below. Cardinality here denotes the distinct number of values that we can have in our system for a particular type of *entity*. There can be a scenario where we are storing the record for a genuine user who might at best use three distinct devices in his lifetime. In this case, which could also be treated as the best-case scenario, the cardinality would be three for the device id. In another scenario, we might be storing a record for a fraudulent user who is part of a wider syndicate and might have access to a larger number of compromised devices. In this case, which could also be treated as the worst-case scenario, the cardinality would be hundreds of device ids.

All relevant entities, such as 'user_id' and 'merchant_id,' either individually or in combination ('from_id' and 'merchant_id'), can be included in the key. The rationale behind this approach is the dynamic nature of interactions during a user's lifetime on the transaction platform. Due to the platform's extensive user base, there can be numerous users and merchants, leading to a wide array of payment scenarios involving various combinations of 'from id' and 'merchant id.'

Similarly, in the system, there can be a substantial number of keys associated with user ids, given the volume of user data. However, when it comes to performing aggregations based on specific combinations, such as the user_id and device_id or the user_id and ip, it becomes crucial to accommodate the device_id or ip within the key. This decision is driven by the need to counter potential fraudulent activities. In such cases, a user might attempt to manipulate their ip or utilize multiple devices to exploit the system. By incorporating the device_id or ip in the key, these fraudulent behaviors can be more effectively identified and mitigated.

Failure to include the device_id or ip in the key could result in significant drawbacks. Specifically, it could lead to inflated record sizes and higher latencies when these

aggregated data points are stored in the value map. This could impact system performance and responsiveness, which is undesirable for efficient data processing.

Conversely, the same approach is not necessary for currency and transaction status data because they have fixed values. These attributes remain consistent and do not require inclusion in the key for aggregation purposes. By focusing on the dynamic aspects of the data (such as 'from_id' and 'merchant_id' combinations) in the key, the system can optimize the latency incurred (keeping record sizes small) by efficiently storing fixed-value attributes like currency and transaction status in value map.

Simultaneously, it is essential to note that storing the device_id and ip in the record key is not mandatory. As demonstrated in the example provided to explain the methodology, the decision to include these attributes in the key may vary depending on the specific use case. It ultimately falls to the operator to determine the cardinality based on the system's requirements and the specific aggregation scenario they are addressing.

While writing the schema if we want to maintain aggregates for *entities* which might be repetitive, we can reuse the same nested map to store the aggregates at different nested levels. If we want aggregates for all the possible values of device id and status, all the possible values of device id, status and currency and all possible values of device id we can rewrite the same nested map as shown in Figure 5f, which is much smaller in size as compared to the one written earlier in Figure 5d.

With the schema described above, we can reduce the total number of keys stored in RAM from one billion and two hundred million keys to approximately two hundred million keys for a merchant fraud engine according to one or more embodiments. This new architecture may allow most of the data to be moved from the more expensive volatile memory to the lesser expensive non-volatile memory (volatile memory utilisation goes from ~80 GB to ~12.8 GB). At the same time, it gives us the flexibility

to control the latency of record retrieval by selectively controlling the number of network I/O operations that have to be performed at the time of retrieval.

,

There could also be a scenario where we might have to aggregate for user ID, and merchant ID and status and in this case, since the best case cardinality for merchant id would be a couple of thousands (any transaction platform operator would have a lot of merchants), a new *“recordKeyEntities”* label would be created containing user id and merchant id, whereas status would be put inside the *“aggregates”* label.

Once a transaction is received from different Kafka streams, the same schema described above can be used to store the information in the Aerospike Database as well. For each entity, we define a path from which the entity’s value needs to be extracted from the payload received in the Kafka stream. All these values are then extracted and temporarily stored in volatile memory. Then we parse the given schema and as we encounter different entities under the *“recordKeyEntities”* and *“entities”* labels, we obtain their values from the map stored in volatile memory and upsert their values in the actual record by using *“Aerospikes’ operate”* command.

For example, we can use the following sample map shown in Figure 5g to store the paths from which the value for the entities needs to be extracted from the payload received in the Kafka stream.

The aforementioned map is then used to create a new map where each entity and its corresponding value in the current payload is stored in volatile memory temporarily. That map can look like as shown in Figure 5h.

obtained from a Kafka stream payload that looks like as shown in Figure 5i.

We parse the schema, iterating over each entry under the *“sets”* label. Each entry under the *“configs”* label leads to creation of a new record in an Aerospike set

whose name would be derived by combining the values present in “recordKeyEntities” label. If we are parsing the above schema, and we consider the first entry under the “sets” label, then it signifies that 3 Aerospike Sets need to be created namely risk\$from.1, risk\$from.2, risk\$from.3. If we take in consideration just risk\$from.1 then the record key would be fetched by first fetching the mapping for `from` label from the map depicted in Figure 5i. In this case it is ‘FromId’. Subsequently, the value of ‘FromId’ is obtained from the map. So, for the set record key would be ‘103182985’.

For the value map corresponding to record key ‘103182985’, schema under label `1` in the configs map would be parsed and the schema would be obtained which would look like as shown in Figure 5j.

This would in turn be then parsed by again first fetching the mapping for each entity and then later obtaining the actual values. Then all these aggregates would be stored in a map where it will be ensured that overlapping entities are written only once. For our case, it would look like as shown in Figure 5k. The aggregates in this map are incremented with the relevant values (for example, count and amount) if the entry for the same map already exists. So, for an aerospike set risk\$from.1 a key-value pair would be stored where key would be ‘103182985’ and value would be as shown in Figure 5k.

Claims

1. A system comprising
 - a communication server;
 - at least one user communication device having an associated user and configured to initiate a transaction which includes transaction data; and
 - communication network equipment configured to establish communication with the communications server, and the at least one user communication device;

wherein the communications server comprises at least one processor(s), an upper tier memory, and a lower tier memory, the server being configured, under control of one or more of the at least one processor(s), to execute instructions stored in the upper tier memory or the lower tier memory to:

 - store a key in the upper tier memory, wherein the key is derived from one or more entity identifiers in the transaction data;
 - store a nested value map or a portion of a nested value map aggregated from the transaction data in the lower tier memory, and
 - determine whether to approve or deny the transaction based on the nested value maps;

wherein the user communication device is further configured to indicate to the user whether the transaction has been approved or denied.
2. The system of claim 1 wherein the upper-tier memory is volatile memory, and the lower-tier memory is non-volatile memory.
3. The system of claim 2 wherein the volatile memory is Random Access Memory (RAM), and the non-volatile memory is one or more Solid State Drives (SSD).

4. The system of any preceding claim wherein the one or more entity identifiers are selected from the group consisting of user id, IP address, device id, merchant id, and any combination thereof.
5. The system of claim 4 wherein the key is selectively stored in the upper-tier memory or the lower-tier memory depending on a first predetermined criteria.
6. The system of claim 5 wherein the first predetermined criteria is whether a cardinality of the key is below a first threshold, if the cardinality of the key is above the first threshold, the key is stored in the upper tier memory.
7. The system of claim 5 or 6 wherein the cardinality of the key is modelled based on a worst-case scenario of a predicted behaviour of the user.
8. The system of any preceding claim further comprising determining whether to divide at least one nested value map or a portion of a nested value map stored in the lower tier memory into two or more record sets depending on a second predetermined criteria.
9. The system of claim 8 wherein the second predetermined criteria is whether the size of the nested value map or the portion of a nested value map is below a second threshold, if the size is above the second threshold the nested value map or the portion of a nested value map is stored in the new record set.
10. The system of claim 9 wherein the second threshold is 500KB.
11. The system of any preceding claim wherein at least one nested value map or portion of a nested value map is reused to store an aggregated value at different nested levels.

12. A method performed in a communication server apparatus for a transaction platform provider, the method comprising, under control of a processor of the communication server apparatus:

storing a key in an upper tier memory, wherein the key is derived from one or more entity identifiers in transaction data received by the transaction platform provider; and

storing a nested value map or a portion of a nested value map aggregated from transaction data in a lower tier memory.

13. A computer program or computer program product comprising instructions for implementing the method of claim 12.

14. A tangible non-transitory storage medium, storing instructions, which when executed by a processor, causes the processor to perform the method of claim 12.

15. A communication server apparatus for a transaction platform provider, the communication server comprising at least one processor, the communication server apparatus being configured, under control of one or more of the at least one processors, to execute instructions, to:

store a key in an upper tier memory, wherein the key is derived from one or more entity identifiers in transaction data received by the transaction platform provider; and

store a nested value map or a portion of a nested value map aggregated from the transaction data in a lower tier memory.

16. A user communication device for a transaction platform provider, the user communication device comprising a processor and a memory, the user communication device being configured, under control of the processor, to execute instructions stored in the memory, to:

initiate a transaction which includes transaction data with a server;

receive a signal indicating whether the transaction has been approved or denied from the server, based on a plurality of nested value maps stored on the server, a key being stored in an upper tier memory of the server, wherein the key is derived from one or more entity identifiers in the transaction data and the nested value maps are aggregated from the transaction data, the nested value maps being stored in a lower tier memory of the server; and

indicate to the user whether the transaction has been approved or denied.

17. A communication server apparatus for a transaction platform provider, the communication server comprising a processor and a memory, the communication server apparatus being configured, under control of the processor, to execute instructions stored in the memory, to:

receive raw data regarding a plurality of transactions for a plurality of users;

store the raw data in a time-series database;

generate aggregated data based on the raw data and store the aggregated data in a not only Structured Query Language (NoSQL) database;

determine a plurality of risk parameters for a given transaction and/or the user based on at least the aggregated data;

decide whether to approve or deny the given transaction based on the risk parameters; and

if the transaction is approved, send the transaction details to a payment service provider to process the transaction.