FlureeDB, A Practical Decentralized Database

Brian M. Platz
Andrew “Flip” Filipowski
Kevin Doubleday

November 7, 2017

Abstract

Applications are the core value drivers of technology, and at the foundation of almost every application is a database. Decentralized applications are no different in their need for a database, however a practical decentralized database did not exist before FlureeDB. A primary obstacle is *consensus*, which comes with high cost and high latency, making a full crypto-database impractical for the needs of applications that require fast response times for an appropriate user experience, transactional volumes, and a host of other enterprise requirements.

FlureeDB provides a solution wherein a crypto-database can be segmented into amorphous parts, each part having different levels of blockchain consensus that align with the requirements of the respective data set. FlureeDB’s query engine allows joins across multiple databases, enabling a single system to consist of a hybrid of consensus characteristics. A database can thus realize speed and privacy where needed, and transparency and trust where that is the desired characteristic. FlureeDB takes full advantage of its immutable blockchain core to offer compelling features not found in a traditional relational or document databases including time-travel, full audit trails and detectable modification attempts.

Table of Contents

1 Introduction

FlureeDB was built as a comprehensive database for both internal and external use. When used externally, FlureeDB is an ideal solution for multiple organizations or individuals to share a common set of data in a decentralized manner with trust and visibility. Examples include cross-organization identity management, crowd-managed data sets, shared financial accounts, or supply-chain management. When used internally, FlureeDB can store internal data with extremely fast transaction times while still retaining most of the blockchain benefits. Fluree uniquely allows both internal and external databases to be merged within query, allowing apps to talk to a single database regardless of existing internally, externally, or both.

FlureeDB compares to traditional databases in the following ways:

1. **Decentralized** - A FlureeDB database can run under a set of predefined rules in a manner where no entity controls it.

2. **Fault tolerance and censorship resistance** - Due to its decentralized nature, FlureeDB guarantees maximum uptime and no possibility for data regulation.

3. **Immutability** – Most databases are update-in-place and don't capture time as a first-class aspect of its data organization. Once a backup window has expired, historical context is lost forever. FlureeDB is an immutable blockchain, all historical context is present and unalterable.

4. **Horizontal Scaling Query** - By separating the query engine from the blockchain transactor, query can scale near linearly. The tradeoff is eventual consistently, which Fluree compensates with sync-point specifications.

5. **DB Compatibility -** The versatile underlying Flake format at the core of FlureeDB makes materializing data into various query patterns simultaneously fairly simple, thus allowing FlureeDB to act as both a Graph database and Document database in the current

implementation, with others possible.

6. **Time Travel** - Both of Fluree's current reference query engines (Graph database and Document database) allow any query to be issued at any point in time in history (as-of any block). It means long-running batch processes never have to worry about data changing underneath them, and the database can be passed around or inspected as an immutable object.

7. **Partitioning** - Every database on FlureeDB can be segmented into one or more partitions, and each supports atomic transactions within the partition. This means a single traditional database table can be split into potentially billions of partitions, each partition having different consensus properties, permissions, and data. This characteristic enables highly scalable writes.

8. **Cross-database joins** – Most databases are considered a single isolated entity. FlureeDB recognizes that certain parts of a database system may have different consensus requirements. Some parts may be used only inside an organization to power business applications, whereas other parts may be shared with external organizations where trust and visibility is paramount. FlureeDB is designed to easily enable multiple databases to act as a single system through query, joining data across them.

9. **Built-in Permissions -** A flexible permission model is built into FlureeDB and solves complex, relational permission needs down to an entity + attribute level, making the database suitable for a broad set of business applications. These same permissions control the data that can be modified by different users (token holders), enabling it to exist in a decentralized manner where it abides by predefined rules.

10. **Data Expiration -** When transacting, every flake can have a unique expiration time. The database will automatically drop flakes from query when that time occurs. A merkle root for multiple expirations within a single block ensure integrity of the block hashes even if some data has passed expiration.

11. **Automatic data subscriptions** - Fluree can introspect queries and know exactly what data changes will affect any given query. An optional JavaScript library is available to keep web-based user interfaces up-to-date automatically.

2 FlureeDB Novelty in Approach

FlureeDB is a protocol token whose blockchain stores a series of point-in-time facts as atomic units we call Flakes. Every Flake is — and forever will be — unique. Flakes can be thought of as events asserting (or retracting) a fact at a particular point in time. A block is a collection of Flakes as a single atomic transaction, and represents each state change of the blockchain.

FlureeDB's primary characteristic needs to be scalability, and therefore sharding is core to the blockchain. Every database is a separate blockchain, and groups of databases are placed into shards. The number of shards can grow over time as demand increases. Block/transaction time is also important, but FlureeDB gives the option of running in different consensus modes when block time is the primary driver of a given set of data. Internal consensus can achieve transaction times in the 10s of milliseconds. Broad consensus will always take extra time.

Fluree uses two tokens, a primary token (Primary Token) and work token (Work Token). The Work Token has relatively static prices for database operations roughly proportional to the computing and storage costs they require. The Primary Token can be exchanged with other users, and when transacting, an exchange rate of Primary Token to Work Token is specified. The two-token approach is similar to that of Ethereum.

Tokens can be held by individual accounts, or by the databases themselves. When submitting a transaction, a user will spend their own tokens unless the database has established rules that allow it to spend tokens that it holds. FlureeDB allows a rich permission model that can control exactly what pieces of data can be modified by whom and also which transactions must be paid for by the transacting user.

3 Elementary Components

FlureeDB has the following components that make up the entire system:

1. Fluree Blockchain – a central blockchain responsible for handling currency transactions, registering database names, and storing proof-of-storage information from mining pools.
2. Fluree Primary Token – The cryptocurrency in which payments are made (token name to change).
3. Fluree Work Token - The Token used when performing database transactions, or database functions.
4. Fluree Wallet – A unique identifier that can hold tokens, can be used in database permissions, and can be used as an encryption key.
5. Mining Pools – A consortium of miners that guarantee certain levels of consensus.
6. Fluree Database – A registration record of a database name and a schema to enforce the data within.
7. Database Functions – A set of functions that participate in atomic transactions which have access to the most current state of a database partition.
8. Database Users - Users registered within the database that have access to certain permissions.

4 Storage

A key aspect of any database is storage. FlureeDB allows the public blockchain to not only set rates for storing data, but gives the user the ability to specify an expiration for every piece of data. While the blockchain metadata never expires in order to maintain the integrity of the chain, the various ‘facts’ (Flakes) asserted in each block can expire at any time specified. Therefore, storage rates can vary based on timeliness of the data.

When transacting data that expires, an additional merkle root will be included in the block metadata. The merkle root contains each of the varying expiration times (in epoch milliseconds), along with a hash of only the flakes with that expiration time. The result of this merkle tree is the final block hash. This allows any miner to be challenged with any as-of time to produce flakes that result in the hash, assuming the as-of time has not yet expired. If the miner being challenged feels the as-of time requested has expired, it can respond accordingly.

Storage rate algorithms are an item for research, however the premise is that short-term storage will cost more and decrease with time. A storage fee is paid when transacting the data based on the expiration (or lack of expiration). The residual storage fees are paid to miners over time, according to the storage rate algorithm.

In order to allow for adjusting prices, a voting mechanism for storage price at a future date (i.e. 30+ days out) can be placed. The price is always the average of those next n days. The date should be set as far out as possible to avoid sudden price changes, but not so far as to be out of touch with the cost of storage.

5 Database Transactions

Transactions are applied to a database (or database partition) as a collection of hashmaps that each convey changes to a particular entity. A simple example of a transaction that is modifying a product's name and price would be (in JSON):

```
[{
  "_id":  ["product/id", "widget100"],
  "name": "A widget in size 100",
  "price": 90.95
}]
```

(Code Sample 1)

Assuming the product currently has a price of 100.00 and a name of "A widget," this transaction would return a result that looks like:

```
{
  "tempids": {},
  "block":   55,
  "hash":
"b8a17bfe6dc9b9de1d2f90be0eeaf0fc6104838161ae8d6be1470105e6a62062",
  "flakes":
  [["product:123", "product/name", "A widget", 55, false, 0],
   ["product:123", "product/price", 100.00, 55, false, 0],
   ["product:123", "product/name", "A widget in size 100", 55,
true, 0],
   ["product:123", "product/price", 99.95, 55, true, 0],
   ["block:55", "_block/hash",
"b8a17bfe6dc9b9de1d2f90be0eeaf0fc6104838161ae8d6be1470105e6a62062",
55, true, 0],
   ["block:55", "_block/prevHash",
"7defabe24295dd859b098da0ca38d55c2416fc8bf83f0122cdfa80bc0b21402d",
55, true, 0],
   ["block:55", "_block/instant", 1509545167655, 55, true, 0],
   ["block:55", "_block/tx",
"cec8d655b3a071ec16f5b4a7636fc765a7deb80415cd650c7359debea320d0c0",
55, true, 0],
   ["block:55", "_block/user", 435546567345, 55, true, 0]]
}
```

(Code Sample 2)

Note the Flakes with 'false' assertions: these are the facts that are no longer true as of this block. Assuming all nodes providing consensus have the same data set, they will always have the identical set of Flakes as the result of a transaction.

Transaction results contains the following keys:

Key	Description
tempids	A mapping of all tempids for new entities to their final resolved entity id
block	The block number for this transaction: every database starts at block 1 and increments. If you want to query the database as of this block, ignoring any future changes, simply supply this block number (55) with any query. To query the database immediately before this block, issue a query and specify block 54.
hash	The final hash-value for this block. The hash is computed by applying a consistent sort order to all

	transaction Flakes except "_block/hash", serializing via JSON utilizing UTF-8 encoding, and performing a hash (currently SHA3-256). If differing expiration times are present, a merkle root will also be included allowing the hash to be verified as-of any point in time that may exclude expired Flakes.
Flakes	All the Flake tuples that are part of this block. See Flakes Section for a description of each tuple position. For clarity here, the entities (product:123 and block:55) are represented as strings instead of their 64-bit integer, and attributes (i.e. product/price) similarly as a string instead of their corresponding 32-bit integer.

(Table 1)

This example transaction in code sample 1 contained the special key: "_id". There are currently three special keys that may be used in each transaction map and all beginning with an underscore ("_") to reflect that they are not attributes:

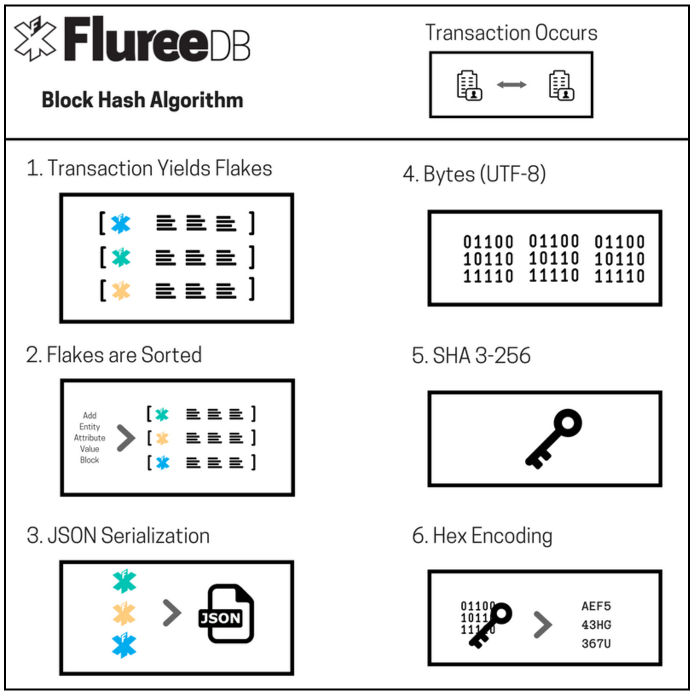
Special Key	Description
_id	Required. Any of the three methods of uniquely identifying an entity: 1. Entity ID (64-bit integer), which is automatically created for every new entity 2. Any unique attribute and its value (as code example 1 demonstrates). The schema definition for the attribute must indicate it is unique for it be used as a valid _id. 3. A temporary-id for new entities. The final entity ID generated (or resolved) will be returned with the block transaction data in the 'tempids' key of the result.
_action	Optional. Explicitly state action to apply. If not provided, the action will be inferred if possible. Valid values are: insert, upsert, update or delete.

_exp	An optional expiration time to use for new Flakes added in the block. Note any Flake retraction (delete) will always use the same expiration time of the original Flake assertion (insert), regardless of the _exp time provided.

(Table 2)

6 Block metadata

Every transaction results in a collection of Flakes that represents the state change in the database. That collection, along with some block metadata (also represented as Flakes), make up the entirety of a block. The block metadata includes the previous block's hash, and once the collection of Flakes is properly sorted and serialized, a new hash is created and stored for the current block, locking in the immutability and the block's chain.



The following block attributes are stored with every transaction:

Block Attribute	Description
_block/hash	The final hash for all of the properly sorted and serialized Flakes that occurred within a single block transaction. If no Flakes expire, this is the result of

	the Flake hash. If Flakes expire, this is the result of the merkle tree stored in _block/expHash.
_block/expHash	If expirations are used for data, this stores the merkle root for each expiration time along with affected flakes. The result of this merkle tree is the _block/hash.
_block/prevHash	The _block/hash of the prior block. By hashing this value into the current block, it forms the immutable blockchain.
_block/user	A reference to the user (wallet) who issued the transaction that created this block.
_block/tx	A reference to the transaction receipt for currency used to process this block.
_block/instant	A timestamp in epoch milliseconds from the miner that processed this transaction. This can be queried to quickly locate the latest block as-of a point in wall clock time, according to the miner. This instant can never be less than the prior block.
_block/userInstant	An optional user supplied instant in epoch milliseconds, which allows a different way of querying as-of blocks via a user timestamp.

(Table 3)

7 Flakes

The building block of FlureeDB is a time-ordered collection of facts, called Flakes, that are grouped into ACID compliant transactions and collectively form a blockchain block. Block metadata is included with each transaction which includes, amongst other things, the prior block's hash and the current block hash. The inclusion of these hashes creates the blockchain that cannot be modified without detection. Block metadata is also structured as Flakes, which enables it to be queryable like any other database data.

Flakes are simply tuples that consist of entity, attribute, value, block-id, assertion (true/false) and expiration time. This simple format can easily be materialized into various query engines which scale horizontally. This enables various database types to be expressed from this core set of atomic data elements, with support for both a graph database and document database. Each are optimized for their respective strengths and each instantly offer time-travel and the ability to issue any query as of any block (time) in history.

7.1 Flake format

A Flake consists of six elements as referenced in **Table 4**. Depending on the language or access circumstance, it can readily be represented as a six-tuple or a map. Every Flake will be unique within a database / partition.

Name	Key	Position	Description
Entity	e	0	A unique 64-bit integer identifier. An entity is roughly equivalent to a relational database row.
Attribute	a	1	The attribute represented as a 32-bit integer. An attribute is roughly equivalent to a relational database column. Each attribute must be defined in the containing stream's schema. The attribute name is an alias to the ID used here, and within transactions or queries either can be used interchangeably. Because an attribute name is an alias to an ID, the underlying ID can be changed if needed (i.e. an incompatible schema change), making it ideal to use attribute names in your end-user applications.
Value	v	2	The attribute's value.
Block	b	3	A reference to the block that asserted this Flake, which allows all associated metadata from that block to be easily

			retrieved.
Add / Retract	add	4	A Boolean value indicating if this Flake was added (true) or retracted (false) from the database
Exp. epoch millis	exp	5	Optional expiration time represented as epoch milliseconds, of when this Flake is no longer relevant. Query engines should drop this Flake at this time.

(Table 4)

8 Schema

All streams (which are somewhat analogous to a relational database table) and attributes (which are somewhat analogous to a relational database column) must be defined in the schema. New attributes and streams can be added any time. Some schema modifications are allowed to existing streams or attributes where integrity of the existing data can be maintained (i.e. changing a single-cardinality value to multi-cardinality, or changing an instant type to a long integer). Importantly, schemas are enforced for new transactions, and therefore only the latest schema is used for this validation. Prior versions of the schema can be queried like any other data, as-of any point in time historically.

Schemas are created/modified with normal database transactions, as the data that defines the schema is itself just a set of Flakes within the database. A simple transaction example to add a new stream for storing product data, and several attributes might look like:

```
[{
 "_id":     ["_stream", -1],
 "name":    "product",
 "doc":     "A stream to hold product data",
 "version": "1"
},{
 "_id":   ["_attribute", -1],
 "name":  "product/id",
 "doc":   "The product's unique identifier",
 "type":  "_attribute.type/string",
 "unique": true
},{
 "_id":   ["_attribute", -2],
 "name":  "product/name",
 "doc":   "The product's name",
 "type":  "_attribute.type/string",
 "index": true
},{
 "_id":    ["_attribute", -3],
 "name":   "product/price",
 "doc":    "The product's price",
 "type":   "_attribute.type/float",
 "index": true
}]
```

(Code Sample 3)

This transaction establishes a new stream called “product” and three new attributes, “product/id”, “product/name” and “product/price.” The “product/id” is a string, and marked unique, which the database will enforce uniqueness. This also allows product/id to be used to uniquely refer to an entity, so it can be used as an _id in queries or subsequent transactions.

Both attributes “product/name” and “product/price” are labeled as indexed. They can thus be used in ‘where’ clauses to find corresponding entities, in addition to range scans (i.e. all products with price > 50 and < 100). All attributes labeled as “unique” are also indexed, so in this example all three attributes will be indexed. However only “product/id” will be guaranteed to exist for only one entity.

8.1 Streams

Streams are roughly equivalent to a traditional relational database table. Entities are always placed in a stream, and the stream ID itself gets encoded into the high order bits of the entity’s id. This ensures that all entities within a single stream have data locality and help avoid O(log n) for pulling index data from persistent storage.

In order to store the core operational data, the database needs to run itself (schema, users, permissions) -- a bootstrapped schema is installed as the genesis block of every new database. All of these pre-installed streams and attributes are prefixed with ‘_’ to help distinguish them from end-user schema items. The base streams installed in the genesis block and their purpose is highlighted in **Table 5.**

Stream	Purpose
_block	Block metadata
_stream	Stream (aka table) schema definitions.
_attribute	Attribute (aka column) schema definitions.
_tag	Tags are a special attribute type that get automatically generated and resolved. Perfect for use as enum values, for linking disparate entities together, or just 'tagging' entities.
_user	Core user table for those with query/transaction privileges within the database.
_auth	Auth records, linked to a _user as a component. Enables logins, key verification, etc. An auth record links to one or more roles which define permissions. This allows differing permissions for a single user depending on the method they used to authorize.
_role	Roles group a set of permissions for querying and transacting. A single user can have many roles (via _auth records), and roles can give explicit access to streams or attributes, or supply a predicate function that dynamically determines access rights.

(Table 5)

8.2 Attributes

While streams group common types of entities together, attributes describe each entity. By default, any attribute can be applied to any entity. In this way Fluree shares more in common with a columnar/document (NoSQL) database than a relational database. Entities within a stream can be restricted to only allow certain attributes by configuring an optional 'spec'. Attributes must have a namespace, and by convention the namespace is the stream name the attribute is intended to be used for. i.e.

for the attribute product/id in the previous example, “product” is the namespace and “id” is the name, and its primary purpose is to be used within the ‘product’ stream (although not required).

Available attribute options are described in the following table:

Attribute	Description
_attribute/name	Unique attribute name. Must have a namespace and name components, separated by “/”. (string)
_attribute/doc	Optional doc string about the attribute’s purpose. (string)
_attribute/type	The attribute’s value type. (tag)
_attribute/unique	True if the database should enforce uniqueness for this attribute. (boolean)
_attribute/multi	True if this attribute is multi-cardinality and can hold multiple values. (boolean)
_attribute/index	True if the values should be indexed. (boolean)
_attribute/upsert	True if we allow automatic upserts for the entity if the unique constraint finds an existing match. Only applicable for _attribute/unique attributes. (boolean)
_attribute/component	True if this attribute holds sub-components that should be automatically resolved with the parent and automatically deleted if the parent is deleted. (i.e. a user’s addresses should be deleted if the user is deleted). This only applies to attributes whose _attribute/type is “_attribute.type/ref” (a reference to another entity… like a SQL join). (boolean)
_attribute/noHistory	True if the history should not be retained for

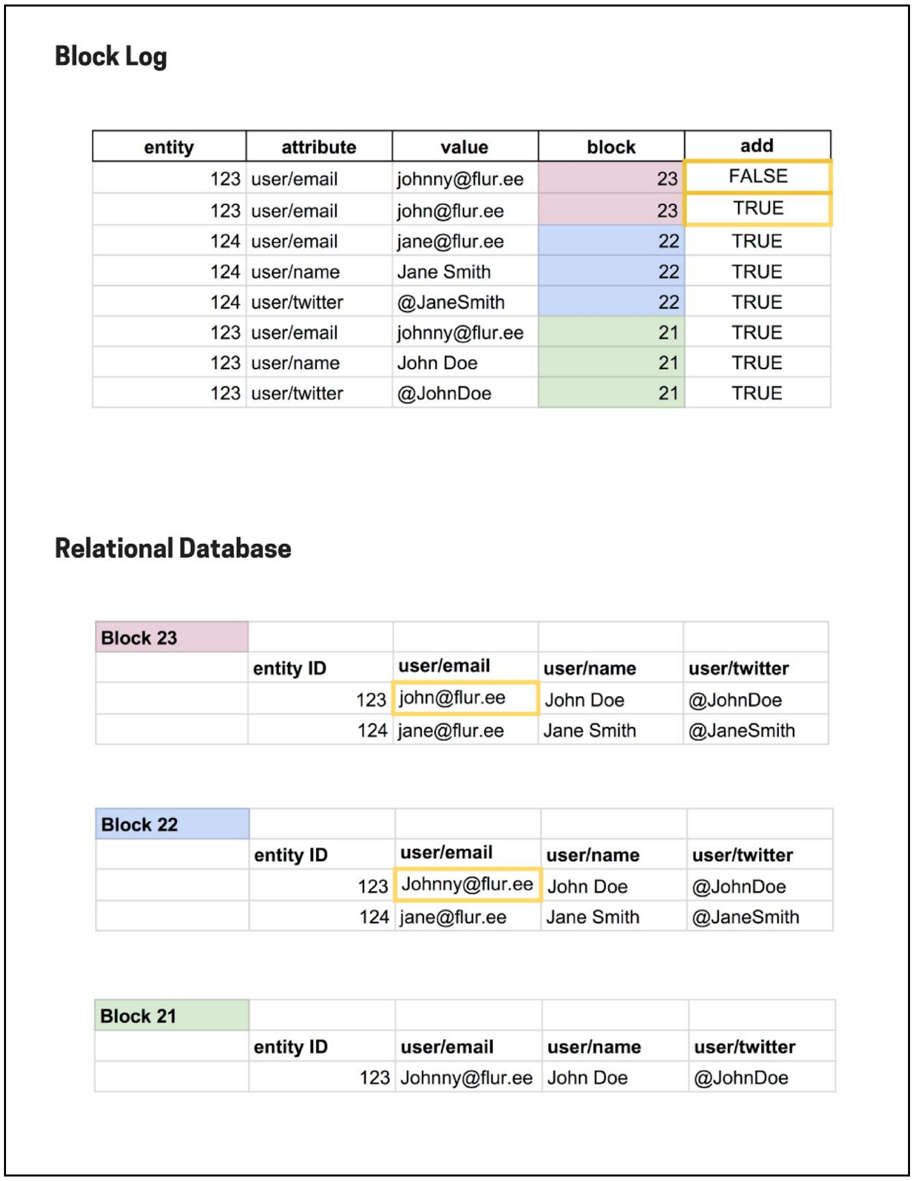
	changes to this attribute. This only applies to FlureeDB running in private consensus mode, as a public blockchain cannot drop history. (boolean)
_attribute/restrictStream	Only applies to reference types (where type is "_attribute.type/ref"). By default a reference type can refer to any entity. By specifying restrictStream, only references to entities in the specified stream will be allowed. This becomes important for the GraphQL query interface which relies heavily on types, and without this a generalized schema will not fully work in GraphQL. (string)
_attribute/spec	An option specification that further restricts the allowed values beyond type. Specs are a set of predicate functions that can ensure, for example, a string type matches a regular expression, or a number type falls between some range of acceptable values. The predicate functions are expandable, and more will be made available over time as needed. (JSON)
_attribute/encrypted	An optional boolean value that indicates if the value of this data being stored is encrypted. If so, type assertions will be ignored and the database expects the transaction value to be a string. Query engines with the proper decryption key will need to validate decrypted values and handle exceptions accordingly. (Boolean)

(Table 6)

9 Query

FlureeDB's query engine is where all the components of FlureeDB come together to provide tremendous utility. Query engines read deltas from the transactor(s) they are monitoring, which can be internal or external blockchains and can horizontally scale to meet any demand needs.

Query engines amount to a materialized view of Flakes. However the reference implementations provide many capabilities not found in traditional databases. Currently, we've built a powerful graph database and a document database using this method. Both allow historical point-in-time queries, rich view permission models, database functions and more. Many applications can leverage the built-in permissions and database functions to power an app without requiring an application tier. A diagram of flakes from blocks and their virtual materialization into different database versions is referenced in Figure 2 below:



Block Log

entity	attribute	value	block	add
123	user/email	johnny@flur.ee	23	FALSE
123	user/email	john@flur.ee	23	TRUE
124	user/email	jane@flur.ee	22	TRUE
124	user/name	Jane Smith	22	TRUE
124	user/twitter	@JaneSmith	22	TRUE
123	user/email	johnny@flur.ee	21	TRUE
123	user/name	John Doe	21	TRUE
123	user/twitter	@JohnDoe	21	TRUE

Relational Database

Block 23				
	entity ID	user/email	user/name	user/twitter
	123	john@flur.ee	John Doe	@JohnDoe
	124	jane@flur.ee	Jane Smith	@JaneSmith

Block 22				
	entity ID	user/email	user/name	user/twitter
	123	Johnny@flur.ee	John Doe	@JohnDoe
	124	jane@flur.ee	Jane Smith	@JaneSmith

Block 21				
	entity ID	user/email	user/name	user/twitter
	123	Johnny@flur.ee	John Doe	@JohnDoe

To extend the utility of the query engine, we've also built a reference JavaScript library that can be embedded in applications to proxy all queries and manage real-time data synchronization - essentially it acts as an embedded FlureeDB. The synchronization is similar in capability to that of the application libraries of Meteor or Apollo GraphQL subscriptions, however it is automatic and extremely granular. Developers do not need to do anything extra -- every app they build can update in real-time automatically and, also, offer 'rewind' using Fluree's point-in-time queries.

Each database can establish how it chooses to be materialized - into either a GraphDB or DocumentDB. While not currently supported, there is nothing preventing it from being materialized into multiple DB types.

9.1 Graph Database

Fluree's Graph database offers robust capability including unlimited recursion, rich query, and database functions. This capability is provided with quick response times by keeping 'hot' data in memory. FlureeDB will automatically swap out data that has not been accessed recently to keep current queries extremely fast.

Indexes are updated and shared across Graph databases using persistent functional data structures. To avoid updating indexes on every new block, each index also holds a queue of recent blocks, and can defer updating indexes until some queue size threshold has been reached.

A storage protocol for indexes allows back-end storage to be swapped according to the needs of the database. Storage can be done locally on disk, or leverage some variation of a distributed file system. The Graph database never updates index segments, but rather always writes new segments when there is an affecting change when flushing a new index. It borrows this characteristic from Bigtable-style databases like Apache Cassandra.

9.2 Document Database

Fluree's Document database offers fairly traditional document database capability in addition to time travel. The document database has less index requirements and differs from the Graph database in that it doesn’t store flakes directly with index segments in memory.

Most queries can leverage fast disk-based storage, and therefore enable extremely large datasets with still very good query capability, however lacking some of the flexibility that a GraphDB offers.

FlureeDB