Clover: Property Graph based metadata management service

Miloš Simić
Faculty of Technical Sciences, Novi Sad, Serbia
milos.simic@uns.ac.rs

***Abstract*—As the file systems continue to grow, metadata search is becoming increasingly important way to access and manage files. Applications are capable to generate huge amount of files and metadata about various things. *Simple metadata* (e.g., file size, name, permission mode), has been well recorded and used in current systems. However, only limited amount of *metadata*, which not record only attributes of entities but also relationships between them, are captured in current systems. Collecting, processing and querying such large amount of files and metadata is challenge in current systems. This paper present Clover, a metadata management service that unifies files/folders, tags, relationships between them and metadata into generic property graph. Service can also be extended with new entities and metadata, by allowing users to add their own of nodes, properties and relationships. This approach allow not only simple operations such as directory traversal and permission validation, but also fast querying large amount of files and metadata by name, size, date created, tags etc. or any other metadata provided by users.**

I. INTRODUCTION

The continuous increase of data stored in the cloud, storage systems, enterprise systems etc. is changing the way we search and access data. Compared to the various database solutions, including the traditional SQL databases [1] and the NoSQL databases [2-4], file systems usually shine in providing better scalability(i.e. larger volume and higher parallel I/O performance). They also provides better flexibility (i.e. supporting both structured and unstructured, as well as non-fixed data schemas). Therefore, a large fraction of existing applications are still using file systems to access raw data. However, with large volumes of complex datasets, the decades-old hierarchical file system namespace concept [5] is starting to show the impact of aging, falling short of managing such complex datasets in an efficient manner, especially when these data comes with some simple metadata. In other words, organizing files (data) in the directory hierarchy can only be effective and efficient for the file lookup requests that are well aligned with the existing hierarchical structures. For today's highly variable data a pre-defined directory structure can hardly foresee, let alone satisfy the ad-hoc queries that are likely to emerge [17]. Metadata can contain user-defined attributes and flexible relationships. Metadata describes detailed information's about different entities like files, folders, users, tags etc. and relationships between them. These information's extend simple metadata which contains attributes from individual entity and basic relationships, to more detail level. Current file systems are not well-suited for search because today's metadata resemble those designed over four decades ago, when file systems contained orders of magnitude fewer files and basic navigation was enough [5]. Metadata searches can require brute-force traversal, which is not practical at large scale. To fix this problem, metadata search is implemented with separate search application, with separate database for metadata as in Linux (locate utility), Apple's Spotlight [6], and appliances like Google [7] or Kazeon [8] enterprise search. This approach have been effective for personal use or small servers, but they face problems in larger scale. These applications often require significant disk, memory and CPU resources to manage larger systems using same techniques. Also these applications must track metadata changes in file system, which is not easy task. Existing storage systems capture simple metadata to organize files and control file access. Systems like Spyglass [10] and Magellan [11] have also been proposed as tools to store and manage these kinds of metadata. While collecting metadata current systems still lack a mechanism to store, process and query such metadata fast. At least some challenges like Storage System Pressure, Effective Processing/Querying, and Metadata Integration should be addressed [12]. The problem with approaches done in the past is that they relied on relational [1] or key-value [14] databases to store and unify metadata. There have been studies that try to fix inefficiency of retrieving and/or managing files, by offering search functionalities from desktop and enterprise systems. For these environments, returning consistent file search results in real-time or near real-time becomes a necessity, which in and by itself is a challenging goal. This paper propose unifying all metadata into one property graph while files remain on file system. All applications and services can store and access metadata by using graph storage and graph query APIs. With this in mind, all applications and services store data on the file system in the same way, and we can further improve performance using optimization techniques for storing data. Complex queries can express easier as graph traversal instead of a join operation in relation databases. Using graph to represent metadata we also gain rapidly-evolving graph techniques to provide better access, speed, and distributed processing.

This paper is organized as follows. Section II present graph model for metadata. Section III present related work. Section IV present system design and implementation, also show used tools. Section V show experimental results. Section VI summarize conclusions and briefly propose ideas for future work.

II. GRAPH-BASED MODEL

Researches already consider metadata as a graph. The traditional directory-based file management generate a

tree structure with metadata stored in *inodes* [15]. Also file system is designed as tree structure. These trees are graphs enriched with annotations that provide more information's. The metadata graph is derived from the *property graph model* [16] (Figure 1), which have vertices (nodes) that represent entities in the system, edges that show their relationships, and properties that annotate both edges and vertices that can store arbitrary information's that user what. These information's are usually stored as properties of vertices or edge in form of *key-value* pair that are usually separated with ':' for example `name: clover, size: 12kb` and so on. **Usually it's not necessary that all vertices or edges contains same set of properties or key-value pairs.**

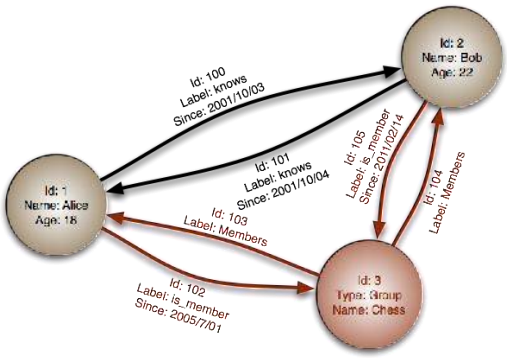

Figure 1. Property graph [28]

A. *Vertices to edges*

Clover define three basic types of vertices, as follow:

Files: represents file on file system, does not contain file content

Folders: represents folder on file system that can contain other folders or files

Tags: represent small metadata information that group other files and/or folders by some (user defined) text. Tags makes filtering easier.

Also users can define their own entities trough APIs for example users or administrators and later on can know which user created some file or folder.

B. *Relationships to edges*

Relationships between entities represent same relationships in file system, and carrying the same semantic. Every file can be *child* of every folder, also every folder can be *child* of every folder in file system structure. Every edge is directed relationship *from child to parent.* Also, relationship between files/folders and tags exist on logic level, and it is not necessarily stored in file system data.

Users are free to add their own relationships and enrich the semantics between data. For example *create* relationship can be added and we can know which user create some edge.

C. *Properties*

Vertices and their relationships have annotations on them. In graph model these annotations are stored as *properties.* These properties are attached to vertices and/or relationships in key-value pars. There is none predefined properties for vertices or relationships, and user add them. Limitation is, that key of every property must be unique in every node/relationship. It can be added more restrictive rule that values for every relationship/edge must be unique like in relation database. Users can always extend model with new properties and enrich semantic of model.

Properties are usually used to select or query specific edges and relationships from others. Examples are name: clover, type: python and so on.

III. RELATED WORK

There is dozens of solutions that have been proposed to fix the inadequacy of file systems in fast file retrieval and filtering, to some extent. These solutions can be broadly divided into the three categories [17]:

File search engines, which rely on the crawling process to catch up with new updates periodically, are unlikely to keep the file index always up-to-date [10-12]. Because of its nature of periodically updates these kind of systems can lead to inaccurate retrieval results. None of the existing file-search engines is designed for large-scale systems. Some of these engines are Apple Spotlight [6], Google Desktop search [7], Microsoft Search [8].

Database-based metadata services use databases as a additional metadata service running on top of file systems. These database-based metadata service have the same limitations like every database-based [2, 3] storage solution. Their performance could not match the I/O workloads on file systems [10, 11]. Also, SQL schema is static and it is not suitable for the exploratory and ad-hoc nature of many big data and HPC analytics activities [18, 19].

Searchable file system interfaces provide file search functions directly through the file systems. Research prototypes that attempt to provide such interfaces include HPC rich metadata management systems [13], Semantic File System [20], HAC [21] and WinFS [22], VSFS [17]. All of these systems serve end-user's needs for retrieving files which means that they will try to find the files based on the keywords provided by users, and have very limited support for the metadata query [10, 23]. These queries might not be useful for analytics applications that rely on range queries or multidimensional queries to fetch the desired data. Furthermore, similar to the file-search engines, these systems perform parsing within the systems, which limits the flexibility in handling the high variety of the datasets.

IV. DESIGN AND IMPLEMENTATION

Clover is composed of few parts. Main part is *Cover service* which handles all HTTP requests from clients, and response to them. Also, this service do all communication to storage infrastructure and metadata service.

Clover service understand all basic commands on files/folders that are common on every file system and operating system. Supported commands are: create (folders only), rename, copy, move, remove, list. With this

approach, clover is released from any scheduled tasks to update metadata, and potentially show not consistent state. On every command intended to the storage infrastructure, metadata is updated. With this in mind, clover can use some current and/or future algorithms to improve speed of these operations. All of these operations are done through Python modules.

In additions to these, Clover provides tagging, search, filter operations on metadata storage. When files/folders are opened, their rate is updated. If search provide more results than single item, the higher rated files/folders are on top of the list as top hit. Figure 2 show Clover architecture.

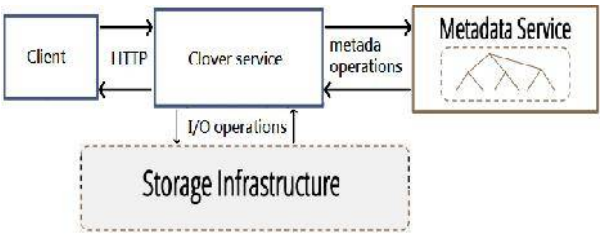

Figure 2. Clover architecture

Metadata service is implemented using graph database [24]. This database store all metadata in form of edges, relationships and properties for every item in storage. Service is also able store information's that exists only on logical level, like who created element, who send item, where is download from etc. and enrich metadata and provide more semantics to it. With this in mind much powerful search can be provided.

Vertices contains at least: name and path to files/folders on storage infrastructure. **Path must be unique**, so *unique* constraint is added to every file/folder vertices. It is recommended, that vertices and/or relationships contains also date created, date modified, last accessed date for better querying, but it is not necessary.

Users are free to extend these properties. Every vertices and relationship, or group of them, can be *labeled* with some free text and make search even sassier and simpler. Metadata service labeled every file with *FILE*, folder with *FOLDER* and tag with *TAG* label to logically distinguish these items, and make querying a lot easier and faster. When file/folder is child of some other folder, that relationship is created and labeled with *CHILD* label. This relationship should have at least *since* property to describe since when files/folders are children of that specific folder. Files and/or folders that are that are tagged are connected to tag vertices over *TAGGED* labeled relationship. Recommendation for *since* property is applied here as well.

Metadata service must provide fast search mechanism, and indexing. Labels are mechanism to relatively fast filter items. This might be acceptable in some cases but if we're going to be looking up some fields frequently, then we'll see better performance if we create an index on that property for label that contains that property. Users can add their own indexes trough clover service APIs. Metadata service provide indexes on *name* property, assuming that file name is used mostly in searching.

Why graph database and not relational database? A graph database... is an online database management system with CRUD methods that expose a graph data model [18]. Two important properties:

- Native graph storage engine: written from the ground up to manage graph data
- Native graph processing, including index-free adjacency to facilitate traversals

The problem with relational approach are joins. All joins are executed **every time** when query is executed, and executing a join means to search for key in another table. With indices executing a join means to lookup a key, B-Tree index speed is O (log (n)).

Graph databases are designed to: store inter-connected data, make it easy to evolve database and to make sense of that data. Enable extreme-performance operations for discovery of connected data patterns, relatedness queries greater than depth 1 and relatedness queries of arbitrary length. People usually use them when have problems with join performance, continuously evolving data set (often involves wide and sparse tables) or the shape of the domain is naturally a graph (like file system).

Early adopters of these databases were Google: Knowledge graph [25], Facebook: Graph search [26]. It show's it is easy to use, it is really fast, and users can query almost on their natural language.

A. Neo4j

Neo4j [27] is used as the database for storing metadata. Neo4j is open source, it has largest ecosystem of graph enthusiast, community is large 1000000+ downloads 150+ enterprise subscription customers including 50+ global 2000 companies (January 2016). Most mature product is in development since 2000, in 24/7 production since 2003.

This database show good connected query performance. Query response time (QRT) [28] is given by formula (1)

```
QRT=f(graph density, graph size, query
            degree)        (1)
```

Graph density is average number of relationships per node

Graph size is total number of nodes in the graph

Query degree is number of hops in ones query

RDBMS has exponential slowdown as each factor increases, and Neo4j performances remains constant as graph size increases. Performance slowdown is linear or batter as density and degree increase.

Neo4j using pointers instead of lookups and doing all joining on creation of vertices and relationships. Also contains *profiler* embedded, and we can detect bottle necks and fix them. Figure 3 show comparison of RDBMS and Neo4j.

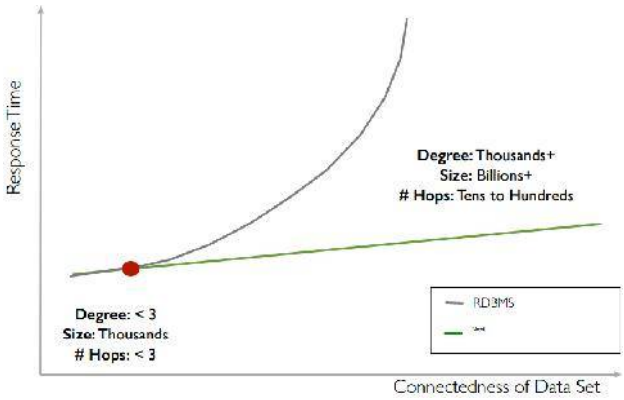

Figure 3. Comparison RDBMS vs Neo4j [28]

Neo4j uses *Cypher* [29] query language. This language can easily mapped graph labels on natural language and make querying a lot easier. For example, if we have two nodes *A, B* and both of them contains name property, and one relationship between them labeled with *LOVES.* Simple query to figure out *friends who likes pie*

```
START me = (p:PEOPLE{name:'me'}),
        pie = (t:THING{name:'pie'})
MATCH me-[:FRIEND]-> (friends:PEOPLE),
        friends -[:LIKES]->pie
RETURN people
```

V. EXPERIMENTAL RESULTS

In order to evaluate the performance system presented in Sections 4, search engine was implemented in Python and Neo4j version 2.3.2 for windows. All experiments were made using a 2 GHz Pentium 4 workstation with 4 GB of memory running Windows 7 and Linux. For dataset, Python27 folder is crawled containing 15084 nodes, 15083 relationships and deep recursive structure of sub files and/or folders. No attempts have been made to optimize Java VM (java version "1.8.0_71", SE build 1.8.0_71-b15,), the queries etc.

Experiments were run on Neo4j and MySQL out of the box with natural syntax for queries. The graph data set was loaded both into MySQL and Neo4j. In MySQL a single table.

Figure 4 show comparison results on MySQL, Neo4J, locate and Windows search when searching folder by given name stoneware in *Python27* directory.

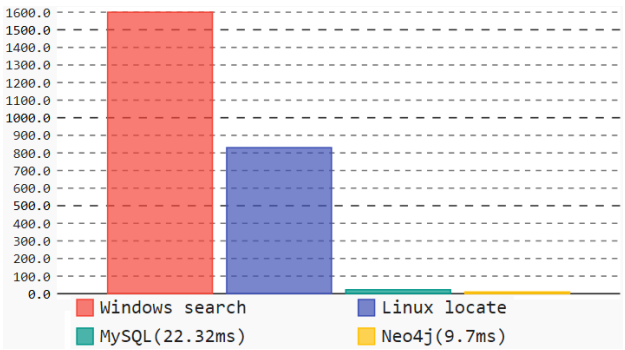

Figure 4. locate, Windows search, MySQL, Neo4J comparison searching folder by given name

Figure 5 show comparison results on MySQL, Neo4j and locate command when searching for child nodes that have **.py* extension of folder by given name stoneware in *Python27* directory.

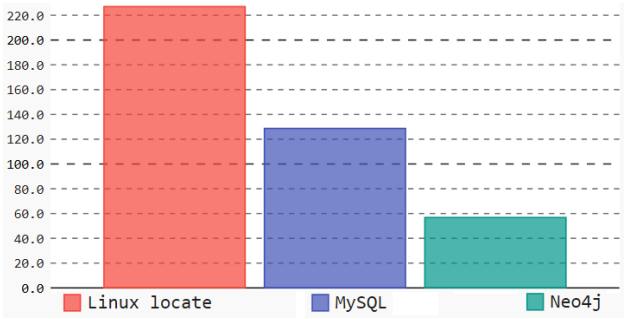

Figure 5. locate, MySQL, Neo4J comparison on searching child nodes that contains *.py as extension of given folder

Figure 6 show comparison results on MySQL, Neo4j and locate when retrieving file/folder attributes for given exact file location.

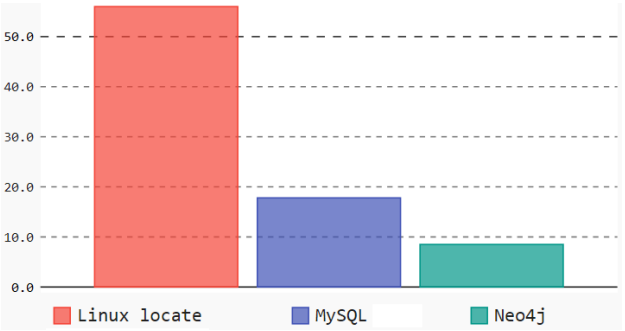

Figure 6. locate, MySQL, Neo4J comparison on retrieving file/folders attributes

Results include time on sending and receiving HTTP requests.

VI. CONCLUSION

As amount of data and files now days become larger and larger, current systems lack to do fast metadata search. This paper present Clover, a graph-based mechanism to store metadata, and search large-scale systems. Clover model data is in form of property graph, where vertices are presented as edges of graph, and they are connected over relationships. Both vertices and relationships contains properties to more describe them, and give them more semantics to them. These properties are stored in key-value form. Inspiration comes from Facebook and Google which use this approach to enable fast search.

There are numerous ways to improve Clover in future. First to add role-based access control (RBAC) to separate which users can access which files. Second, to improve search by adding *Domain Specific Language* specifically designed for natural language. This will make search even easier. Third, content of text files can be stored in some document database so Clover can search inside content of files. Forth, Clover can be extended with framework to support big-data. Fifth, all operations that

affect storage are currently synchronous. Future work should enable asynchronous operations for every function on file system. This can be handy especially with bigger files and operations that takes a lot of time to be executed (copying or moving big amount of files etc.). Also system should be tested on server configuration with larger amount of files/folders and different kind of not just simple, but also rich metadata by giving more semantic relationships.

REFERENCES

[1] Oracle database. http://www.oracle.com/us/products/database/overview/index.html, accessed 2016.

[2] K. Banker. MongoDB in Action. Manning Publications Co., Greenwich,CT, USA, 2011.

[3] D. Borthakur and et al. Apache hadoop goes realtime at facebook.SIGMOD '11. ACM

[4] G. DeCandia and et al. Dynamo: amazon's highly available key-value store. SOSP '07

[5] Daley R., Neumann P., A general-purpose file system for secondary storage. In *Proceedings of the Fall Joint Computer Conference, Part I* (1965), pp 213-229

[6] Apple Spotlight, https://support.apple.com/en-us/HT204014, accessed 2016

[7] Google inc. Google enterprise, https://www.google.com/work/, accessed 2016

[8] Kazeon, Kazeon search enterprise, http://www.emc.com/domains/kazeon/index.htm, accessed 2016

[9] Microsoft. Windows Search. https://support.microsoft.com/en-us/kb/940157, accessed 2016

[10] A. W. Leung, M. Shao, T. Bisson, S.Pasupathy, E. L. Miller, Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems, in *FAST*, vol 9, 2009, pp. 153-166

[11] A. Leung, I. Adams, E. L. Miller, Magellan: A Searchable Metadata Architecture for Large-Scale File Systems, University of California, Santa Cruz, Tech. Rep. UCSC-SSRC-09-07, 2009.

[12] L. Xu, H. Jiang, L. Tian, and Z. Huang. Propeller: A scalable real-time file-search service in distributed systems. ICDCS '14

[13] D. Dai., R B. Ross, P Carns, D. Kimpe, Y. Chen, Using Property Graphs for Rich Metadata Management in HPC Systems

[14] S. C. Jones, C. R. Strong, A. Parker-Wood, A. Holloway, D. D. Long, Easing the Burdens of HPC File Management, in Proceedings of the sixth workshop on Parallel Data Storage. ACM, 2011, pp. 25-30

[15] J. C. Mogul, Representing Informatinos about Files, Ph.D dissertation, Citeseer, 1986, Texas Tech University

[16] Property Graph, https://www.w3.org/community/propertygraphs/, 2016

[17] L. Xu, Z. Huang, H. Jiang, L. Tian, D. Swanson, VSFS: A Searchable Distributed File System, parallel data storage workshop, 2014

[18] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: the twitter experience. SIGKDD Explor. Newsl., 14(2):6–19, Apr. 2013

[19] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. SIGMOD '09

[20] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In SOSP '91, 1991

[21] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In OSDI '99.

[22] Microsoft. WinFS: Windows Future Storage. http://en.wikipedia.org/ wiki/WinFS, accesed 2016

[23] Y. Hua and et al. Smartstore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In SC '09.

[24] I. Robinson, J. Webber, E. Eifrem, Graph databases, O'Reilly, 2015, ISBN: 9781491930892

[25] Google Knowledge graph, http://www.google.com/intl/es419/insidesearch/features/search/knowledge.html, accessed 2016

[26] Facebook graph search, https://www.facebook.com/graphsearcher/, accessed 2016

[27] Neo4j, http://neo4j.com/, accessed 2016.

[28] Goto conference 2014, http://gotocon.com/dl/goto-chicago-2014/slides/MaxDeMarzi_AddressingTheBigDataChallengeWithAGraph.pdf, accessed 2016

[29] Cypher, http://neo4j.com/docs/stable/cypher-introduction.html, accessed 2016