Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL -

Zhen Hua Liu, Dieter Gawlick

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065, USA

{zhen.liu, dieter.gawlick}@oracle.com

1. ABSTRACT

RDBMSs are designed to manage well-structured data requiring users to design a schema before storing and querying data. This is the 'schema first, data later' approach. However, there are significant amount of unstructured data and semi-structured data that cannot be effectively modelled this way. Even if certain parts of the data can be modelled using schema, the inclusion of all fields would typically lead to a very large schema with many optional fields and with frequent schema evolution as data instances vary widely and evolve fast. Obviously, these data requires the 'data first, schema later/never' approach. We call these data **Flexible Schema Data (FSD)**. In this paper, we describe the engineering principles and practices to manage FSD in RDBMSs to meet FSD's unique requirements and challenges. We describe the limitations and issues of current practices and potential research opportunities. Having a single data platform for managing both well-structured data and FSD is beneficial to users; this approach reduces significantly integration, migration, development, maintenance, and operational issues.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *Relational databases, transaction processing.*

General Terms

Algorithms, Management, Performance, Design, Languages, Standardization.

Keywords

JSON, SQL/JSON, Schema-less, No-SQL, XML, SQL/XML, Flexible Schema, MongoDB

2. INTRODUCTION

The focus on the 'schema first, data later' approach has so far prevented RDBMSs from being the ideal platform of managing FSD. Instead, FSD like support has been implemented in specialized DBMSs due to RDBMS is found to be inadequate to support schema evolution [43] to handle data whose structure changes a lot over time.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

For managing unstructured documents, it is common to use content management systems that store documents as files with text index providing keyword search [13, 16]. For managing document-oriented semi-structured data, such as XML, MarkLogic NoSQL system [34] is popular with XQuery as query language. As JSON becomes the data-centric semi-structured data format, MongoDB [33] based NoSQL systems with JSON specific query language become a popular choice for managing JSON data. Polygolt storage with NoSQL [40] is getting popular. As the volume of FSD grows at an ever faster rate, the trend towards using these specialized NoSQL [4] based database systems accelerates. This trend leads to significantly increased complexity of the management of data since users cannot manage all of their data in one platform. When users have to work with different data platforms, they have to write data integration code in their applications. Users can not query all of their data using a single high level declarative query language. Instead, they have to use different query languages for querying different data and implement their own join algorithms to join between relational data and FSD. Last, but not the least, many specialized systems lack essential advanced functionality, such as bi-temporality, provenance, and fine grain security that are standard in modern RDBMSs.

In contrary to [2], our goal is to enable RDBMSs to manage FSD along with relational data and thereby leveraging all the advanced data management services that have been developed over many years for relational data. All leading RDBMS platforms have supported XML data management using the SQL/XML [10] query language during the last decade so that XML and relational data can be queried and managed together. Lately, Oracle [21], Vertica [24], TeraData [25], Postgres SQL [7], and Sinew Hadapt System [22] are all supporting JSON data management by extending SQL so that JSON and relational data can be queried and managed together in an RDBMS. The benefits of extending RDBMSs to manage FSD are:

- Enabling schema-less data application development paradigm (data first, schema later/never) in RDBMS.
- Enable agile style rapid data access with maximum schema flexibility (Schema on Read but not on Write Paradigm) in RDBMS.
- Efficient consolidated single data management platform – covering both relational data and FSD to reduce integration issues, simplify operations, and eliminate migration issues.

- Efficient productive declarative application development – by leveraging SQL as a set-oriented query language to declaratively query domain specific FSD.

We understand the rationale of "One Size Does Not Fit All" [31] argument as a way to encourage out-of-box thinking and re-architecting RDBMSs to handle a variety of new challenging data management requirements that do not fit the original relational data management paradigm that has been established more than four decades ago. However, it is desirable to present a single system which hides the complexity of multiple architectures instead of having users to manage multiple systems. Therefore, we prefer to build an evolutionary path for extending RDBMSs to support FSD data management. *Nevertheless, since FSD management has challenged the fundamental assumption of RDBMSs that require existence of schema to store, index and query data, we do need new ways of thinking how to store, query, update and index FSD differently from relational data.* That is, we need to **think out-of-the-schema**. Indeed, management of FSD challenges us to think how to store, query and index data **without up-front schema definition**?

To accomplish this goal, we leverage the RDBMS extensibility technology for managing user defined object types, functions and indexes [12, 14]. Applying extensibility ideas leads us to the current engineering principles and practises for managing **FSD** in RDBMSs as follows:

- **Storage Principle**: Use the document-object-store model by storing FSD as one object without relying on any static schema & E/R model to decompose FSD into relational tables. That is, no schema on write. Flexible schema that is embedded in FSD can be computed as data-guide to provide schema on read capability.

- **Query and Update Principle**: Leverage SQL as a declarative **S**et-oriented **Q**uery **L**anguage. That is, position "NoSQL" to mean **N**aturally **o**pen **S**et-oriented **Q**uery **L**anguage to embed FSD domain specific query language. FSD domain language provides query and navigation capability for both schema and data for each FSD instance.

- **Index Principle**: Index FSD using relational table index and search index. The relational table index derivable from data-guide and query workload provides efficient relational access for pre-defined query access patterns. The search index using generalized inverted index strategy provides efficient search for ad-hoc query access patterns.

The main contribution of this paper is a detailed analysis and discussions of these principles to understand the rationale of why we propose to use these principles to manage FSD, the issues and limitations when practising these principles, and the potential new research challenges and opportunities to manage FSD equally well as that of relational data in an integrated RDBMS platform. Although database extensibility technology is well-known as RDBMS engineering practise [12,14], we found that abstracting this engineering practise using these three principles helps to more adequately address the challenges of FSD.

Outline of the Paper: Section 3, 4, 5 goes into details for storing, querying, updating and indexing FSD respectively with section 6 on advanced data management capability for FSD. Section 7 draws conclusion followed by the acknowledgements in section 8.

3. Storing FSD

3.1 FSD Storage Requirements

The relational design leverages the E/R model [5] which provides a clean separation between structure and data. This method has been very successful for a large class of applications by **extracting common structures out of data as schema.** Schemata are managed by RDBMSs in a central dictionary. Therefore, in the E/R model, a schema has to be defined before data can be loaded.

A collection of FSD data, such as JSON objects, XML documents, has typically a small number of common attributes complemented by a large variety of non-common attributes. The attributes form hierarchical structural relationships. The structure is not easily separable from data content because the structure varies greatly from instance to instance. Shredding FSD collections relationally results in a large number of tables joined by a large number of primary/foreign key relationships and still many tables have many sparsely populated columns [1, 3]. Furthermore, constant schema evolution is required as new sparse attributes are detected in new FSD instances or single occurrence of an existing attribute is detected to have multiple occurrences in new FSD instances. Therefore, this is **not** a **scalable** solution. Instead, the instance schema is embedded in each FSD instance so that each FSD instance is self-contained and can be distributed to different tiers. Schemata of a FSD collection are not managed as central dictionary data but rather computable dynamically as data-guide [19] from all FSD instances stored in a FSD collection.

Abstractly, **schema based data** can be defined as a set of data (which is denoted as 'S') that satisfies the following properties: there exists a set of finite size of dimension (which is denoted as 'D') such that every element of S can be expressed as a linear combination of elements from D.

Flexible schema based data is the negation of Schema based data. That is, there does NOT exit a set of finite size of dimension D such that every element of S can be expressed as a linear combination of elements from set D. Intuitively, schema based data can have unbounded number of elements but has a bounded dimensions as schema definition whereas flexible schema based data has unbounded dimensions.

Because schema based data has finite dimensions, therefore, schema based data can be processed by separating the data away from its dimension so that an element in a schema based data set can be expressed by a vector of values, each of which represents the projection of the element in a particular dimension. All the dimensions are known as schema. Flexible schema based data cannot be processed by separating the data away from its dimension. Each element in a flexible schema based data has to keep track of its dimensions and the corresponding value. An element in a flexible schema based data is expressed by a vector of dimension and value (name-value pair). Therefore, flexible schema based data requires store, query and index both schema and data together.

3.2 FSD Storage Current Practises

Self-contained Document-object-store model: The current practice for storing FSD is to store FSD instances in a FSD collection using document-object-store model where both structure and data are stored together for each FSD instance so that it is **self-descriptive** without relying on a central schema dictionary. New structures can be added on a per-record basis without dealing with schema evolution. Aggregated storage supports full document-object retrieval efficiently without the cost of querying and stitching pieces of data from multiple relational tables. Each FSD instance can be independently imported, exported, distributed without any schema dependency. Table1 shows DDL to create *resumeDoc_tab* collection of resume XML documents, a *shoppingCar_tab* collection of shopping cart JSON objects. SQL/XML standard defines XML as a built-in datatype in SQL. For upcoming SQL/JSON standard [21], it supports storing JSON in SQL varchar, varbinary, CLOB, BLOB datatype with the new 'IS JSON' check constraint to ensure the data stored in the column is a valid JSON object. Adding a new domain FSD by storing into existing SQL datatype, such as varchar or LOB, without adding a new SQL type allows the new domain FSD to have full data operational completeness capability (Transactions, Replication, Partition, Security, Provenance, Export/Export, Client APIs etc) support with minimal development efforts.

T1	*CREATE TABLE resumeDoc_tab (id number, docEnterDate date, docVerifyDate date, resume* ***XMLType)***
T2	*CREATE TABLE shoppingCar_tab (oid number, shoppingCar* ***BLOB check (shoppingCar IS JSON))***

Table 1 – Document-Object-Store Table Examples

Data-Guide as soft Schema: The data-guide can be computed from FSD collections to understand the complete structures of the data which helps to form queries over FSD collection. That is, FSD management with data-guide supports the paradigm of "*storage without schema but query with schema*". For common top-level scalar attributes that exist in all FSD instances of a FSD collection, they can be automatically projected out as virtual columns or flexible table view [21, 22, 24]. For nested master-detail hierarchical structures exist in FSD instances, relational table indexes [11] and materialized views [35], are defined using **FSD_TABLE() table function** (Q4 in Table 2). They can be built as secondary structures on top of the primary hierarchical FSD storage to provide efficient relational view access of FSD. FSD_TABLE() serves as a bridge between FSD data and relational data. They are flexible because they can be created on demand. See section 5.2 for how to manage FSD_TABLE() and virtual columns as indexing or in-memory columnar structures. Furthermore, to ensure data integrity, soft schema can be defined as check constraint as verification mechanism but not storage mechanism.

3.3 FSD Storage Limitations and Research Challenges

Single Hierarchy: The document-object-storage model is essentially a de-normalized storage model with single root hierarchy. When XML support was added into RDBMSs, the IMS hierarchical data model issues were brought up [32]. Fundamentally, the hierarchy storage model re-surfaces the single root hierarchy problem that relational model has resolved successfully. In particular, supporting m-n relationship in one hierarchy is quite awkward. *Therefore, a research challenge is how to resolve single hierarchy problem in document-object-storage mode that satisfies 'data first, structural later' requirement.* Is there an aggregated storage model, other than E/R model, that can support multi-hierarchy access efficiently? Papers [20, 23] have proposed ideas on approaching certain aspects of this problem.

Optimal instance level binary FSD format: The document-object-storage model is essentially a de-normalized storage where master and detail data are stored together as one hierarchical tree structure, therefore, it is feasible to achieve better query performance than with normalized storage at the expense of update. Other than storing FSD instances in textual form, they can also be stored in a compact binary form native to the FSD domain data so that the binary storage format can be used to efficiently process FSD domain specific query language [3, 22]. In particular, since FSD is a hierarchical structure based, the domain language for hierarchical data is path-driven. The underlying native binary storage form of FSD is tree navigation friendly which improves significant performance improvement than text parsing based processing. The challenge in designing the binary storage format of FSD instance is to optimize the format for both query and update. A query friendly format typically uses compact structures to achieve ultra query performance while leaving no room for accommodating update, especially for the delta-update of a FSD instance involving structural change instead of just leaf value change. The current practise is to do full FSD instance update physically even though logically only components of a FSD instance need to be updated. Although typically a FSD instance is of small to medium size, the update may still cause larger transaction log than updating simple relational columns. A command level logging approach [27] can be investigated to see if it is optimal for high frequent delta-update of FSD instances.

Optimal FSD instance size: Although the size of FSD collections can be scaled to very large number, in practise, each FSD instances is of small to medium size instead of single large size. In fact, many vendors have imposed size limit per FSD instance. This is because each FSD instance provides a logical unit for concurrency access control, document and Index update and logging granularity. Supporting single large FSD instance requires RDBMS locking, logging to provide intra-document scalability [43] in addition to the current mature inter-document scalability.

4. Querying and Updating FSD

4.1 FSD Query and Update Requirements

A FSD collection is stored as a table of FSD instances. A FSD instance itself is domain specific and typically has its own domain-specific query language. For FSD of XML documents, the domain-specific query language is XQuery. For FSD of JSON objects, the domain-specific query language is the SQL/JSON path language as described in [21]. Table 2 shows the example of SQL/XML[10] and SQL/JSON[21] queries and

DML statements embedding XQuery and SQL/JSON path language. In general, the domain-specific query language provides the following requirements:

- **Capability of querying and navigating document-object structures declaratively:** A FSD instance is not shredded into tables since hierarchies in a FSD can be flexible and dynamic without being modelled as a fixed master-detail join pattern. Therefore, it is natural to express hierarchical traversal of FSD as path navigation with value predicate constructs in the FSD domain language. The path name can contain a wildcard name match and the path step can be recursive to facilitate exploratory query of the FSD data. For example, capabilities of the wildcard tag name match and recursive descendant tag match in XPath expressions support the notation of navigating structures without knowing the exact names or the exact hierarchy of the structures. See *'.//experience'* XPath expression in Q1 and Q2. Such capability is needed to provide flexibility of writing explorative and discovery queries.

- **Capability of doing full context aware text search declaratively:** FSD instances can be document centric with mixture of textual content and structures. There is a significant amount of full text content in FSD that are subject to full text search. However, unlike plain textual document, FSD has text content that is embedded inside hierarchical structure. Full text search can be further confined within a context identified by path navigation into the FSD instance. Therefore, context aware full text search is needed in FSD domain languages. See XQuery full text search expression in XMLEXISTS() predicate of Q1 and Q2 and path-aware full text search expression in JSON_TEXTCONTAINS() predicate of Q3.

- **Capability of projecting, transforming object component and constructing new document or object**: Unlike relational query results which are tuples of scalar data, results of path navigational queries can be fragments of FSD. New FSD can be constructed by extracting components of existing FSD and combine them through construction and transformation. Therefore, constructing and transforming FSD instances are required in any FSD language. See XQuery constructor expression in the XMLQUERY() function in Q1.

- **Capability of performing component-wise update**: FSD instance shall be updatable at component-wise level. New structure shall be addable to existing structures; existing structures and their values shall be updateable and deletable. XQuery update facility has provided all of these functionalities for XML document. See XQuery update facility expression in XMLQUERY() function in Q2.

4.2 FSD Query and Update Current Practises

While a FSD domain-specific query and update language serves as an **intra-document query language**, SQL can be used as an **inter-document query language**. The current practices of querying FSD is to position SQL as a set-oriented language to provide declarative access of a set of FSD instances by leveraging the set based algebra supported by SQL. By positioning SQL as a **S**et (oriented) **Q**uery **L**anguage, SQL provides the necessary constructs to express set algebra operators, such as selection, projection, join, group by, aggregation, union, intersection and difference among FSD instances. SQL is openable to support a set of FSD_XXX() functions that can embed FSD domain specific query language.

These FSD_XXX() functions are used in strategic places in SQL to filter, process, transform and update FSD instances. See Figure 1 for details.

- **FSD Filtering**: FSD_EXISTS() is used as a conditional expression in a SQL WHERE clause to filter FSD instances.
- **FSD un-nesting**: FSD_TABLE() is used as a table function in SQL FROM clause to unnest collection components within FSD instances into a virtual relational table. Un-nesting can be done recursively, therefore Q4 shows the example NESTED PATH support in JSON_TABLE() to un-nest master-detail-detail relationships. Being concrete form of FSD_TABLE(), XMLTABLE() and JSON_TABLE() are very popular features in RDBMS to provide a relational bridge between hierarchical FSD and flattened relational table. Supporting FSD un-nesting concept can be traced back to SQL over NF2 model [42].
- **FSD Scalar Projection**: FSD_VALUE() is used to extract scalar value within a FSD and then to cast it as SQL built-in type values so that it can be used in a scalar value expression in SELECT, GROUP BY, ORDER BY clauses where scalar values are typically expected.
- **FSD Component Projection and Construction**: FSD_Query() is used to query components within FSD or to construct new FSD in SELECT and UPDATE clause.
- **FSD Update**: FSD_Query() is used at RHS side of UPDATE expression to generate a new FSD instance.
- **SQL JOIN of FSD Tables:** SQL can be used to join multiple FSD tables. This can be accomplished by leveraging the SQL JOIN concept and FSD_VALUE() function. Q6 in table 2 shows the join of *resumeDoc_tab*, *shoppingCar_tab*.

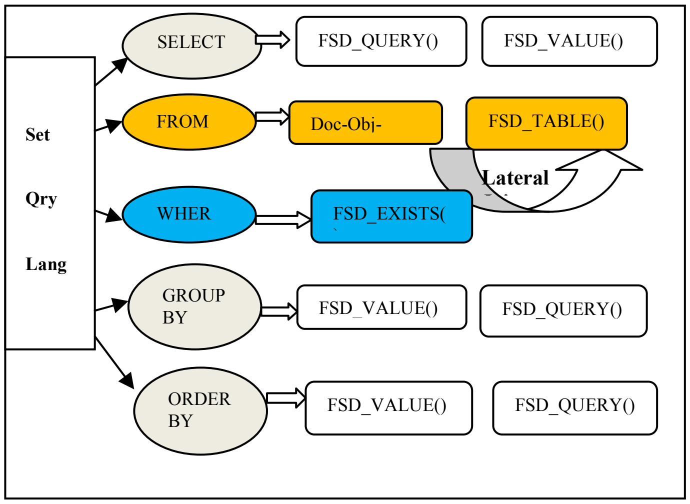

Figure 1 – FSD_XXX() Function Usages in Open-SQL

Q1	*SELECT* ***XMLQUERY****('<summary>{$doc/contact-info, $p//employment}</summary>' PASSING p.resume as "doc") FROM resumeDoc_tab p WHERE* ***XMLEXISTS****('$doc/resume[.//experience contains text "xquery" ftand "json" and .//employmentHistory/employment[starting-time > xs:date("2000-01-01")]] ' PASSING p.resume as "doc")*
Q2	***UPDATE*** *resumeDoc_tab p SET p.resume =* ***XMLQUERY****('copy $new := $doc delete node $new/contact-info/ssn return $new ' PASSING p.resume as "doc") WHERE* ***XMLEXISTS****('$doc/resume[.//experience contains text "xquery" ftand "json" and .//employmentHistory/employment[starting-time > xs:date("2000-01-01")] and .//GPA[. > 3.5]]' PASSING doc.resume)*
Q3	***SELECT JSON_VALUE****('$.shoppingCarDate AS TIMESTAMP) FROM shoppingCarTab WHERE* ***JSON_TEXTCONTAINS****(p.shoppingCar, '$.item.promotion.Description', 'discount and warrenty') and* ***JSON_EXISTS****('$.item?(price > 100 && quantity <=10)')*
Q4	*SELECT p.id, v.itemName, v.itemPrice, v.partName,v.partQuantity, v.partPrice FROM shoppingCarTab p,* ***JSON_TABLE****(p.shoppingCar, '$.items'* ***COLUMNS*** *(itemName varchar(200) PATH '$.itemname', itemPrice number PATH '$.itemPrice', NESTED PATH '$.parts'* ***COLUMNS*** *(partName varchar(100) PATH '$.partname', partQuantity number PATH '$.partQuantity', partPrice number PATH '$.partPrice'))) v*
Q5	*SELECT COUNT(*) FROM resumeDoc_tab p, shoppingCarTab p2 WHERE* ***XMLVALUE****(p.resume, '$p/contact-info/email-address') =* ***JSON_VALUE****(p2.shoppingCar, '$.user.loginName')*
Q6	*SELECT COUNT(*) FROM resumeDoc_tab p, shoppingCarTab p2 WHERE* ***p.resume.contact-info.email-address*** *=* ***p2.shoppingCar.user.loginName***
Q7	***UPDATE*** *shoppingCarTab p2 set* ***p2.shoppingCar.items[1].availability = 'false'*** *WHERE* ***p2.shoppingCarTab.shoppingCarDate*** *BETWEEN TO_TIMESTAMP(:1) and TO_TIMESTAMP(:2)*

Table 2 – SQL/XML and SQL/JSON Example

4.3 FSD Query Limitations and Research Challenges

Syntatic sugar for FSD Domain Language into SQL: The issue for embedding FSD domain language into SQL is that the FSD query appears to be "glued" into SQL instead of being natively part of it. A user friendly syntactic sugar would be to make path navigation appear to be SQL object navigation syntax. Q6 and Q7 show a simplified syntax that makes SQL natively understand the FSD path navigation access. However, an even friendlier SQL FSD language approach shall be investigated to integrate the idea of Schema-Free SQL [30] to query FSD based on data-guide [19] collected from FSD. Modelling JSON and XML path navigation as SQL object type path navigation is very attractive as it provides a uniform language interface to both schema-based SQL99 object type [12,14] and flexible schema based JSON and XML instances.

Declarative Multi-Hierarchy Transformation: To overcome the single hierarchy issue of the FSD storage model discussed in section 3.3, it shall be feasible to come up with a declarative transformation language to transform a collection of FSD instances with one hierarchy to another collection of FSD instances with a different hierarchy having the same semantic equivalence. For example, given a FSD collection of FSD instances, each of which represents a student taking a set of courses, a transformation shall be applicable to generate another FSD collection of FSD instances, each of which represents a course is taken by a set of students. Recall in E-R model, this is handled by maintaining m-n mapping table so that both hierarchies can be generated using SQL. In the document-object-store model, the challenge is to come up with a declarative transformation language extension to SQL to transform hierarchy. Category theory [26] may help us to define transformation algebra between relational model and all of the implied equivalent hierarchical models that can be derived from the relational model so that a path query over hierarchical model has its equivalent SQL query over the relational model. One concrete application of such XPath to SQL transformation is the SQL/XML query rewrite technique that is well-practised in XML enabled RDBMS [41].

DataGude Statistics: All FSD_XXX() functions and FSD_TABLE() can be built into the RDBMS kernel for efficient execution. However, understanding the cost model for FSD accesses and statistics distributions of FSD data-guide are essential for optimizers to get an optimal plan for SQL/FSD query. This issue is presented in paper [22].

Columnar layout of FSD for Vector Set Processing of SQL/FSD Query: The last decade has witnessed drastic performance improvement for relational data via columnar storage and processing [15] and vector processing [28]. The idea has been applied to hierarchical data as nested columnar store [38]. However, Dremel [38] relies on the presence of schema for the nested data in order to construct the columnar storage. Furthermore, record assembly from columnar storage to get original record can be expensive compared with native aggregated store. It is known that relational row-store is good for OLTP workload whereas relational columnar-store is good for OLAP workload. In the same way, FSD binary format at instance level is good for OLTP FSD workload whereas FSD columnar format at set level is good for OLAP FSD workload. Since relational in-memory-columnar structures [29] can optimize both OLTP and OLAP workload by implicitly managing these two dual formats and converting between them on user behalf without forcing users to make up front storage choice, research is needed to apply similar strategy to manage FSD as well. A more attractive approach is to develop an indexing or in-memory columnar layout strategy for FSD. Such FSD columnar layout shall be friendly for vector based processing without relying on any central schema definition.

Single language for both imperative logic and declarative query access for FSD: This has been attempted in OODBMS, Microsoft LINQ and full-fledged XQuery without SQL/XML[43]. However, the challenge is to teach the language compiler and optimizer to understand what is imperative and what is declarative as each of them requires different optimization techniques. This challenge requires integrated research between SIGMOD and SIGPLAN groups.

5. Indexing FSD

5.1 FSD Indexing Requirements

RDBMS indexing techniques, such as B+ tree indices, bitmap indices are defined based on the existence of schema, so are materialized views which is defined based on schema in conjunction with query workload that provide *pre-defined query access pattern*. Therefore, the creation of index and materialized view in RDBMS is based on the paradigm of '*schema first, index definition later*'.

FSD indexing shall be able to provide performance for both pre-defined query access pattern and ad-hoc query access pattern. *Pre-defined query access pattern* in the context of FSD means that users are aware of the partial schema within FSD computed from data-guide so that a relational projection out of FSD in the form of FSD_VALUE() for a set of scalar value projections or in the form of FSD_TABLE() for a set of relational view can be defined. This is referred as **'data first, schema later as index' relational indexing approach.** On the other hand, *Ad-hoc query access pattern* in the context of FSD means users do not have any prior knowledge of the FSD so that a FSD search index is needed to provide efficient evaluation of FSD_EXISTS() with ad-hoc query search. This is referred as '**data first, schema never' search index approach**, which is similar to full text index style search index.

5.2 FSD Indexing Current Practices

5.2.1 Relational Columnar Index for Efficient Relational Column Query Access Pattern Using FSD_VALUE()

In RDBMSs, the result of a SQL function expression over a column is commonly used for range queries, e.g., for range search over an UpperCase() function of a varchar type column, a functional index [12,14] can be created on the result of functional expression over a column to speed up range queries over that functional expression; functional indices can be defined on a FSD column using the FSD_VALUE() function. We propose to use columnar compression techniques [36] to handle result of FSD_VALUE() function. Columnar compression technique clusters all column values of a single column together, therefore, a column having many NULL values and repetitive values is more amendable to columnar compression and results in much smaller size. Therefore, the size of FSD_VALUE() projection values encoded in compressed columnar format is often small enough such that the whole columnar encoded FSD_VALUE() can fit into main memory allowing efficient in-memory scan and vector processing leveraging hardware support, such as SIMD instructions. In contrast to the classical columnar storage usage [15], we promote the idea of using the columnar layout as a secondary indexing structure and not as a primary storage structure. We call this **columnar index or in-memory columnar structure** [29]. The columnar index supports efficient range queries over the columnar projection of FSD_VALUE() and returns a set of DOCIDs, each of which is an ordinal number that identifies a row of the base document-object-store table having FSD that satisfy the range query. Unlike columnar storage, we don't need to stitch columnar data together to get the original FSD since the original FSD can be obtained directly from the primary document-object-store table using the DOCID returned by the columnar index.

To support range queries over multiple scalar values projected from FSD via a set of FSD_VALUE() functions, multiple columnar indexes, each of which maps to a FSD_VALUE() function, can be created. Boolean expression using multiple FSD_VALUE() functions can be processed efficiently by using bitmap merges of DOCIDs from multiple columnar index lookups. Therefore, the columnar index gives us the benefit of both worlds: efficient query processing using columnar compression without the need to stitch to obtain the original row. Indeed, we think that even for processing of pure relational data, *positioning columnar storage structure as an indexing or in-memory columnar structure over the row storage gives us the best of both worlds: fast columnar based search and query without the need to do row stitch.*

5.2.2 Relational Table Index for Efficient Relational View Query Access Pattern Using FSD_TABLE()

FSD_VALUE() can only handle one-to-one scalar projection relationship, not one-to-many master-detail expansion relationship. However, FSD has internal hierarchy representing one-to-many master-detail relationships. For example, both XML and JSON have embedding collection objects in the form of repeating XML elements in XML and JSON array. These collection elements are typically projected out and accessed as relational views defined using FSD_TABLE() by users.

To efficiently process FSD_TABLE() queries, we use the idea of **table index** [11]. Table index can have two physical forms. In a classical row store, table index can internally maintain master-detail relational tables to hold the relational results computed by evaluation of FSD_TABLE(). The master-detail table is linked by internally generated primary foreign key so that the column values in the master table are NOT repeatedly stored in detail tables. Indeed, the table index layout in row store is the same as if FSD were decomposed and stored relationally using E-R design. This physical form is ideal for FSD OLTP workload. However, a more attractive physical form of table index is to leverage the power of columnar index and in-memory columnar structure to load FSD_TABLE() results as in-memory columnar structures without physical materialization. The FSD_TABLE() in memory form can leverage columnar compression techniques to efficiently handle repeated master values and sparse NULL value filled entries so that queries over the in-memory FSD_TABLE() can leverage the full power of in-memory columnar scan and vector processing. This physical form is ideal for FSD OLAP workload with mainly read-only data.

In summary, both the table index and the columnar index are very flexible approaches because they are secondary structures on top of the primary FSD store. Such secondary structures can be dropped and re-created without affecting the base document-object table. Therefore, users have the flexibility to decide what to index based on query workload without the need of changing the base storage. *Although it is not feasible to use the relational-tuple-store model to store FSD due to lack of a central schema to describe every pieces of FSD, it is feasible to use the relational-tuple-store model to define columnar index in the form of FSD_VALUE() and FSD_TABLE() to index FSD when* *the relational query patterns can be extracted out from query workload and used as partial schema to define the index.*

5.2.3 Search Index based on generalized inverted index for Ad-hoc Query Access Pattern Using FSD_EXISTS()

Columnar based table index assume that users know the query pattern and query workload. This is not possible for ad-hoc query use cases. For the document search use case, the path expressions that are used in FSD_EXISTS() may not be known in advance. To handle such an ad-hoc query, a search index over a FSD table without having users to specify what path structures or values need to be indexed is built. In other words, a search index indexes everything in a FSD collection. Search indices can be built based on classical inverted index that indexes all keywords in a document to provide ad-hoc keyword search capability [16]. This provides the basic full text search capability over document centric XML documents or JSON objects. However, unlike classical inverted indices that index only keywords in a document, a **generalized inverted index** is extended to index hierarchical path structures inside FSD to support path-aware full text search scalar range value search workload queries.

Extending inverted index for indexing path structures to handle path-aware full text search: Inverted index is originally designed for full text search [16]. The advantage of inverted indices is that even though they index all keywords of all documents in a document collection, the posting list for each keyword in the inverted index is highly compressed so that the total size of the inverted index is still smaller than the size of original document collection [16]. A smaller index size is obviously I/O friendly since physical disk I/O is generally a primary performance bottleneck in DBMS systems [17]. Each FSD document in a FSD collection indexed by the search index is identified by an ordinal number as a DOCID. The DOCIDs of all documents containing the keyword are stored in a sorted manner with delta-compression within a posting list so that efficient **multi-predicate Zig-Zag pre-sorted merge join (MPPZZSMJ)**, is performed on the posting lists to efficiently handle multi-keywords searches and phrase searches connected by AND, OR, NOT Boolean predicates [17]. By indexing path structures and their hierarchical relationship and leveraging MPPZZSMJ, classical inverted indices can be extended to support efficient processing of path *containment query* and path-aware full text search [18] that is a common query for XML full text search .

<u>Search Index for XML</u>**:** Consider Q1 that has predicate XMLEXISTS() using XQuery full text search to find resumes which has '*xquery*' and '*json*' keywords in their '*experience*' element tag, this XMLEXISTS() predicate can be processed by a generalized inverted index created on *resume* columns. Like classical inverted indices, generalized inverted indices index all keywords so that each distinct keyword (subject to stemming, stop words rules) in XML text node is indexed as an entry in inverted index with the posting list storing not only the DOCIDs of documents containing the keyword but also the positions of the keyword within the document. Such keyword position helps to do phrase search or search a group of keywords within certain distances. Unlike classical inverted indices, the extension in generalized inverted indices is to index all XML tags of XML documents stored in document object table. Each distinct XML tag is indexed and stored as an entry in inverted index with the posting list storing not only the DOCIDs of documents containing the XML tags but also the range of tag open and close positions within the document. These positions are used to test hierarchical relationship for path traverse query and node containing keyword query during the MPPZZSMJ process. The key idea behind extending inverted index to index XML is that it captures both XML path structures (tags and their hierarchical relationships) and content data (full text) together in one index [18]. With such an integrated index, querying structure and data together can be processed efficiently.

<u>Search Index for JSON</u>: In the same way, an inverted index can be extended to index JSON objects stored in JSON collection table as well [21]. Like XML tree nodes, JSON objects and arrays form nested hierarchical relationship that can be indexed using their positions within the JSON object. All JSON object field names are like XML tag names that can be indexed with a posting list containing the DOCIDs of JSON object that contains the object field names and their positions within the JSON object so that JSON_EXISTS() path query can be answered using an inverted index. Full text in JSON object field content can be indexed to facilitate full text query within a JSON path.

Extending inverted index for indexing scalar values to handle path-aware scalar range query search: Compared with XML documents, JSON objects are more data centric. Many JSON object leaf fields are scalar values of numbers, dates and timestamp relational column like datatypes. In classical full text search, it is not capable of processing range predicate query on such scalar data. In XQuery, a range scalar type query is typically mixed with full text query, all of which are searched within a context defined by XPath. Considering Q2, XQuery used in the XMLEXISTS() predicate requires not only full text query search but also range query over scalar data embedded in specific XML element, a date range search and a number range search:

'*.//employmentHistory/employment[starting-time > xs:date("2000-01-01")] and .//GPA[. > 3.5]*'.

Processing such range expressions requires extending the inverted index to auto detect number, date, timestamp scalar values embedded in FSD and then index them. The range-data index structure maps a range of typed data value for a particular datatype to a posting list. The posting list contains the set of DOCIDs of FSD data having that value together with its positions within the document. The MPPZZSMJ processing can be extended to join posting lists for ranged-typed scalar data, textual keywords and path structures using DOCIDs, word positions and range value positions. *The generalized inverted index essentially accomplishes the goal of efficient processing of join-of range data query and full text search query.*

5.3 FSD Indexing Limitations and Research Challenges

Both the columnar index and inverted index share the same property that their compact layout is very query friendly for efficient set processing and updated in batch model. Columnar index and inverted index is not instance update friendly compared with B+ tree index. Columnar store requires in-memory buffering, tuple mover and partition-merging to transform ingestion friendly row format into columnar format. Query over columnar format provides snapshot isolation. Inverted indices [13] are typically asynchronously maintained for large batch of documents and search results are based on snapshot semantics. Providing a real time inverted index has been researched [16]. The solution generally relies on querying the read-friendly indexing structures union with the querying over delta documents that are temporarily in write-friendly structures. It is a research opportunity to integrate columnar index and inverted index layout together so that their presence to users is one index and is always maintained in real time [37]. Furthermore, in the context of RDBMS, the inverted index must provide **transactional consistency between the dual formats:** instance ingestion friendly OLTP format and set query friendly OLAP and search format.

6. Advanced Data Management Support

In addition to the storage, index query and update requirements, advanced data management capabilities developed for relational data, such as: Bi-temporal support [8], provenance, query expression data [9] are equally applicable to the management of FSD data. These advanced data management services in RDBMS provide advanced declarative data services that applications can easily exploit. It would be extremely time consuming and costly to provide these advanced data management services in the application tier.

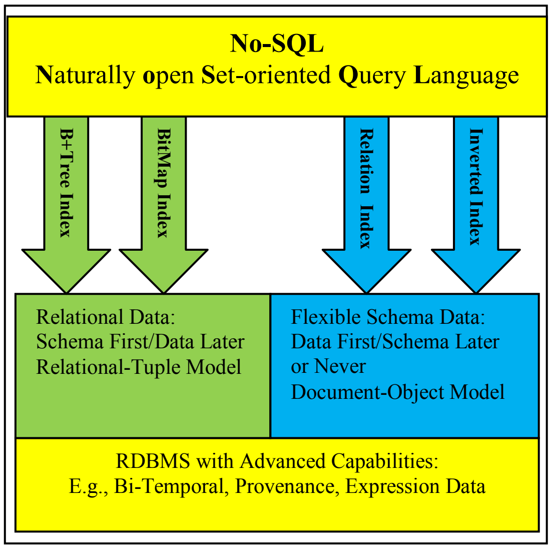

Figure 2 – Unified RDBMS Architecture for supporting both relational data and flexible schema data

In summary, by applying these three principles, Figure 2 shows the unified RDBMS architecture to manage both relational data and FSD. At high level, our storage, query, index principles to manage FSD is similar to the idea of OCTOPUSDB [39]. That is, rather than creating a zoo of DBMS systems with Polyglot persistence [40], we think the primary storage shall be a simple log based document object store with many different storage views maintained as secondary structures: such as index, materialized view or in-memory structures to speed up query and DML workloads from different kinds of usecases. Furthermore, regardless of the choice of primary data storage structures or secondary auxiliary structures to speed up query workload, database application transparency shall be maintained via supporting a declarative query language interface, such as SQL so that the application programs that access data via SQL need NOT be changed. The spirit of storage, query and index principles for FSD can be further applied to big data environment where the primary data can be stored in various forms. SQL extended with domain language remains to be the declarative language to query the data. Intelligent secondary auxiliary structures can be created in-memory to deliver ultra query performance.

7. Conclusion

With the rapid growth of flexible schema data, we are living in interesting time where E.F Codd's relational model [6] that assumes the existence of data schema to manage data is being challenged. A strength of NoSQL systems is the support of the 'data first, schema later/never' approach; i.e., data can be stored without designing a schema first. In this paper, we have shown that RDBMSs can be extended to handle FSD by following three principles: leveraging document-store model, opening SQL with extension functions to declaratively query and update FSD, and adopting relational indexing and inverted indexing strategies. These principles can be implemented in the RDBMS kernels at data storage, query, and indexing layer to make RDBMS a unified platform to manage both kinds of data. The underlying philosophy for the principles is simple: **Treat Schema as if it were data: Store, Index and Query schema along with the data for FSD.** However, there are issues, limitations, and research opportunities to mature FSD data management in RDBMS. In fact, most of the issues do exist in specialized DBMS to manage FSD as well. In contrary to [2], we think that SQL is not dead, relational mode is not dead. Being RDBMS researchers and developers, we shall feel good about living in such exciting time to provide declarative management of data regardless of their existence of schema or not. Let's hope that the year 2015 will be for RDBMS like 1905 for Physics in which the classical framework for a mature field is being challenged to accommodate new phenomenon.

8. Acknowledgements

The authors like to express their thanks to following colleagues: Andreas Behrend, Eric S. Chan, Ronny Fehling, Adel Ghoneimy, Boris Glavic, Oliver Kennedy, and Venkatesh Radhakrishnan, for their constructive suggestions, comments, and critics.

9. REFERENCES

[1] R. Agrawal, A. Somani, Y. Xu: Storage and Querying of E-Commerce Data. VLDB 2001

[2] P. Atzeni, C.S. Jensen, G. Orsi, S. Ram, L. Tanca, R. Torlone: The relational model is dead, SQL is dead, and I don't feel so good myself. SIGMOD Record, 42(2):64-68, 2013

[3] J. L. Beckmann, A. Halverson, R. Krishnamurthy, J. F. Naughton: Extending RDBMS To Support Sparse Datasets Using An Interpreted Attribute Storage Format

[4] R. Cattell: Scalable SQL and NoSQL data stores. SIGMOD Record, 39(4):12-27, 2010.

[5] Chen P.: The Enity-Relationship Model: Toward a Unified View of Data. VLDB 1975: 173

[6] Codd, E. A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13(6): 377-387 (1970)

[7] Postgress SQL for JSON: http://www.postgresql.org/docs/9.3/static/datatype-json.html.

[8] Gawlick, D: Querying the Past, the Present, and the Future. ICDE 2004: 867

[9] A. Yalamanchi, J. Srinivasan, D. Gawlick: Managing Expressions as Data in Relational Database Systems. CIDR 2003.

[10] I.O. for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML)

[11] Z. H. Liu, M. Krishnaprasad, H. J. Chang, V. Arora: XMLTABLE Index - An Efficient Way of Indexing and Querying XML Property Data, ICDE 2007

[12] Z. H. Liu. "Object-Relational Features in Informix Internet Foundation."Informix technical notes. 9.4(Q4 1999):77-95.

[13] G. Salton and M. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, New York, 1983.

[14] Stonebraker,M;, Brown,P; Moore, D. *Object-Relational DBMSs*, Second Edition Morgan Kaufmann 1998

[15] Stonebraker, M.; Abadi, D; Batkin, A.;Chen,X.;Cherniack, M;Ferreira,M.;Lau, E.;Lin,A.;Madden,S; O'Neil, E.;O'Neil,P.;Rasin,A.;Tran,N.;Zdonik,S. *C-Store: A Column-oriented DBMS*. VLDB 2005: 553-564

[16] J. Zobel, A. Moffat: Inverted files for text search engines. ACM Computing. Surveys. 38(2): (2006).

[17] I. Rae, A. Halverson, J. Naughton: In-RDBMS Inverted Indexes revisited. ICDE 2014: 352-363

[18] Z.H.Liu, Y. Lu, H.Chang: Efficient Support of XQuery Full Text in SQL/XML Enabled RDBMS. ICDE 2014: 1132-1143

[19] R. Goldman, J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semi-structured Databases. VLDB 1997: 436-445

[20] H. V. Jagadish, Laks V. S. Lakshmanan, Monica Scannapieco, Divesh Srivastava, Nuwee Wiwatwattana: Colorful XML: One Hierarchy Isn't Enough. SIGMOD Conference 2004: 251-262

[21] Z. H. Liu, B. Christoph Hammerschmidt, D. McMahon: JSON data management: supporting schema-less development in RDBMS. SIGMOD Conference 2014: 1247-1258

[22] D. Tahara, T. Diamond, D. J. Abadi: Sinew: a SQL system for multi-structured data. SIGMOD Conference 2014: 815-826

[23] C. E. Dyreson, S. S. Bhowmick, R. Grapp: Querying virtual hierarchies using virtual prefix-based numbers. SIGMOD Conference 2014: 791-802

[24] Vertica Flex Table: https://my.vertica.com/docs/7.0.x/PDF/HP_Vertica_7.0.x_Flex_Tables.pdf

[25] TeraData JSON data support: http://www.computerweekly.com/news/2240217675/Teradata-Database-15-adds-JSON-addresses-internet-of-things

[26] D. Spivak: http://math.mit.edu/~dspivak/technical_proposalONR--2013.pdf

[27] N. Malviya, A.Weisberg, S.Madden, M. Stonebraker: Rethinking main memory OLTP recovery. ICDE 2014: 604-615

[28] M. Zukowski, P. A. Boncz: From x100 to vectorwise: opportunities, challenges and things most researchers do not think about. SIGMOD Conference 2012: 861-862

[29] http://www.oracle.com/technetwork/database/options/database-in-memory-ds-2210927.pdf

[30] F. Li, T. Pan, H. V. Jagadish: Schema-free SQL. SIGMOD Conference 2014: 1051-1062

[31] Stonebraker,M; Çetintemel, U: "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). ICDE 2005: 2-11

[32] Stonebraker,M.; Hellerstein, J. *What Goes Around Comes Around*. In readings in Database Systems. The MIT Press, 4th edition, 2005.

[33] MongoDB: http://www.mongodb.org/

[34] MarkLogic: http://www.marklogic.com/

[35] I. Elghandour, A. Aboulnaga, D. C. Zilio, C. Zuzarte: Recommending XML physical designs for XML databases. VLDB J. 22(4): 447-470 (2013)

[36] D. J. Abadi, S. Madden, M. Ferreira: Integrating compression and execution in column-oriented database systems. SIGMOD Conference 2006: 671-682

[37] http://www.cs.cornell.edu/bigreddata/publications/2009/usetimpaper.pdf

[38] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis: Dremel: Interactive Analysis of Web-Scale Datasets. PVLDB 3(1): 330-339 (2010)

[39] J. Dittrich, A. Jindal: Towards a One Size Fits All Database Architecture. CIDR 2011: 195-198

[40] P. J. Sadalage, M. Fowler: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence – August 18, 2012

[41] M. Krishnaprasad, Z. H. Liu, A. Manikutty, J. W. Warner, V. Arora, S. Kotsovolos: Query Rewrite for XML in Oracle XML DB. VLDB 2004: 1122-1133

[42] P. Pistor, F. Andersen: Designing a generalized NF2 Model with an SQL-Type Language Interface. VLDB 1986" 278-285

[43] Mohan C: History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. EDBT 2013