Succinct Data Structures and Delta Encoding for Modern Databases

Matthijs van Otterdijk[1,2] and Gavin Mendel-Gleason[1,3] and Kevin Feeney[1,4]

[1] DataChemist Ltd. http://datachemist.com
[2] matthijs@datachemist.com
[3] gavin@datachemist.com
[4] kevin@datachemist.com

December 2, 2019

Modern hardware architectures includes larger main memory and pervasive parallelism. Modern software development processes now incorporate continuous integration/continuous delivery (CI/CD) coupled with version control. These fundamental changes to information technology infrastructure necessitate a re-appraisal of database architecture. TerminusDB makes a radical departure from historical architectures to address these changes. First, we implement a graph database with a strong schema so as to retain both simplicity and generality of design. Second, we implement this graph using succinct immutable data structures which enable more sparing use of main memory resources. Prudent use of memory reduces cache contention while read-only data structures simplify parallel access significantly. Third, we adopted the delta encoding approach to updates as is used in source control systems such as git. This provides transaction processing and updates to our immutable database data structures, recovering standard database management features while also providing in addition the whole suite of revision control features: branch, merge, squash, rollback, blame and time-travel facilitating CI/CD approaches on data.

1 Introduction

There has been an explosion of new database designs, including graph databases, key-value stores, document databases and multi-model databases. Yet the majority of production databases are still based on the RDBMS designs of the 1970s[2].

Meanwhile both hardware infrastructure and software design process have moved on significantly over the course of the last 40 years. In particular machines with terrabytes of RAM are now available for prices reasonable enough for some small and medium sized enterprises.

At the same time flexible revision control systems have revolutionised the software development process. The maintenance of histories, of records of modification and the ability to roll back enables engineers to have confidence in making modificiations collaboratively. This is augmented with important features such as branching, labelling, rebasing, and cloning. When combined with continuous integration/continuous delivery[7][8] (CI/CD) teams of programmers can have confidence that central repositories are maintained in correct states once testing and verification of have been passed.

These two developments suggest a solution at their intersection. Namely the use of in-memory immutable *succinct* data structures and *deltas* as used in revision control systems. TerminusDB demonstrates how these features can be combined to produce a flexible transactional graph database.

2 Design

TerminusDB is a full featured graph database management system (GDBMS) including a rich query language: WOQL (the Web Object Query Language). However, we restrict our attention here to the underlying datastructure design and layout which we have implemented in a Rust[1] library which we call *terminus-store*.

We describe in turn the graph database model which is used, the succinct data structure approach, and finally how we implement revision control type operations using *deltas* which we collect together with some metadata into an object which we term *layers*.

2.1 Graph Databases

Graph databases are one of the fastest growing of the new database paradigms. Since graphs are very general it is possible to render many database modeling techniques in a graph database. The simplicity and generality make it a good candidate for a *general purpose* delta encoded approach to an online transaction processing database.

The TerminusDB infrastucture is based on the *RDF* standard. This standard specifies finite labelled directed graphs which are parameteric in some universe of datatypes. The names for nodes and labels are drawn from a set of IRIs (Internationalized Resource Identifiers). For TerminusDB we have chosen the *XSD* datatypes as our universe of concrete values.

More formally, in TerminusDB a graph G is a set of triples drawn from the set $IRI \times IRI \times (IRI \oplus XSD)$ where IRI is a set of valid IRIs and XSD is the set of valid XSD values. While some RDF databases allow multiplicity of triples (i.e. a bag), the choice of a set simplifies transaction processing in our setting.

For schema design TerminusDB uses the OWL language, with two modifications to make it suitable as a schema language. Namely we dispense with the open world interpretation and insist on the unique name assumption[3]. This provides us with a rich modelling language which can provide constraints on the allowable shapes in the graph.

TerminusDB, following on from the RDF tradition, is not a property graph. However it can model properties using an intermediate nodes and this pattern can be made explicit in the OWL schema design. Again this choice leads to simplicity of the underlying representation, which, as we will see is important when constructing succinct data structures with change sets.

2.2 Succinct Data Structures

Succinct data structures[6] are a family of data structures which are close in size to the information theoretic minimum representation. Technically they can be defined as data structures whose size is:

$$n + o(n)$$

String	Offset	Remainder
Pearl Jam	0	Pearl Jam
Pink Floyd	1	ink Floyd
Pixies	2	xies
The Beatles	0	The Beatles
The Who	4	Who

Table 1: *Plain Front Coding Dictionary*

Where n is the information theoretic minimum size. Succinct representations are generally somewhat more computationally expensive than less compact representations with pointers when working with small problems. However, as the size of the datastructure grows, the ability to avoid new cache reads at various levels of the memory hierarchy (including reading information from disk) means that these representations can prove very speed competitive[5] in practice.

TerminusDB largely borrows its graph data structure design from HDT[9] with some modifications which simplify the use of change sets. The authors originally evaluated HDT as a possibility for a graph which was too large to fit in memory when loaded into postgresql and found that queries on the resulting graph performed very well in practice.

In particular, the primary datastructures of the HDT format are retained, namely *front coded dictionaries*, *bit sequences* and *wavelet trees*.

2.2.1 Plain Front-Coding Dictionary

Due to the unusual quantity of shared prefixes found in RDF data due to the nature of URIs and IRIs, front-coding provides a fast dictionary format with significant compression of data[10].

The primary operations exposed by the datastructure are *string-id* which gives us a natural number corresponding with the string, and *id-string* which gives a string corresponding with a natural number.

The data strucure sorts the strings and allows sharing of prefixes by reference to the number of characters from the preceeding strings which are shared. An example is given in Table 1. The position in the dictionary gives us the implicit natural number identifier.

2.2.2 Succinct Graphs Encoding

Once subject, object and property of an edge have been appropriately converted into integers by use of the subject-object dictionary, the value dictionary and the predicate dictionary, we can use these integers to encode the triples using bit sequences.

Succinct sequences encode sequences drawing from some alphabet σ. In the case of a bit-sequence, $\sigma = \{0, 1\}$. They typically expose (at least) the following primitives:

Triples	Encoding	
$(1,2,3)$	1 2 3	Subject Ids
$(1,2,4)$	1 1 0 1	Encoded Subject Bit Sequence
$(2,3,5)$	2 3 4 5	Predicate Vector
$(2,4,6)$	1 0 1 1 1	Encoded Predicate Bit Sequence
$(3,5,7)$	3 4 5 6 7	Object Vector

Table 2: *Succinct Graph Representation*

- $rank(a, S, i)$ which counts occurances of a in the sequence from $S[0, i]$.
- $select(a, S, i)$ which returns the location of the i-th occurance of a in the sequence S.
- $access(S, i)$ which returns the symbol at $S[i]$.

Given a sorted set of triples, for each subject identifier, in order from $\{0..n\}$ where n is the number of triples, we emit a 1 followed by a 0 for every predicate associated in a triple with that subject. We then produce a vector of all predicates used and the association with the subject is apparent from the position of zeros in the bit sequence.

We repeat the process for predicates and objects resulting in a complete encoded for our triples. We can see an example in Table 2. We have written the vectors in this table so that the triples are vertically aligned, with subjects in blue, predicates in red and objects in green in order to make the encoding easier to see. The subject ids are actually implicit in the number of 1s encoding in the subject bit sequence and are only written in the table for clarity.

This format allows fast lookup of triples based on query modes in which the subject identifier is known, as we can use $select$ to find the position in the predicate vector and subsequently use the predicate identifier to $select$ in the object vector. We use a wavelet tree to enable search starting from the predicate. Details of this can be found in [9].

2.3 Delta Encoded Databases

The use of *delta encoding* in software development is now ubiquitous due to the enormous popularity of the *git* revision control system which makes use of these techniques to store histories of revisions.

Git stores objects which contain either the complete dataset of interest or the information about what is updated (deleted / added) as a delta. All changes to the source control system are thereby simply management problems of these objects.

This approach exposes a number of very powerful operations to sofware developers collaborating on a code base. The operations include:

- **History** Since new updates are actually layered over previous ones, developers can *time travel*, looking into the past, rolling back to the past, or even reapplying changes to current versions.

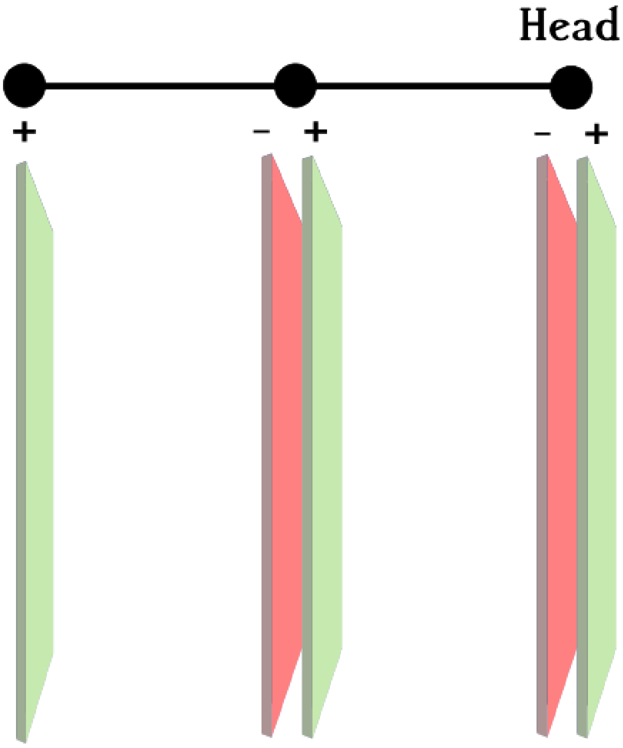

Figure 1: *A graph composed of layers*

- **Branching** Developers can create a derived version of a given code-base where additional operations can be performed without disrupting the original.
- **Merging** When two branches diverge, the changes can be merged into a single version by choosing a strategy for combining changes.

TerminusDB uses an analogous approach to updates. A given database is comprised of *layers* which stands in place of the objecs of git. Each layer has a unique identifier, a 20-byte name. The base layer contains a simple graph represented using the succinct data strucures already described earlier.

Above this layer, we can have further layers. Each additional layer above the base layer is comprised of additional dictionaries for newly added subjects and objects, predicates or values. It also contains the index structures used for the base graph to represent *positive* edges which have been added to the graph. And we have a membership set of *negative* edges which describe those triples which have been deleted as shown in Figure 1.

Each layer has a pointer to the previous layer which is achieved by referring to its 20-byte name.

This immutable chain strucure allows for straightforward uncoordinated multi-read access sometimes called muliversion concurrency control (MVCC) [11] [12]. This approach also makes branching simple to implement. Any number of new layers can point to the same former parent layer.

In order to manage these layers as datastores, we use a *label*. A label is a name which points to one of

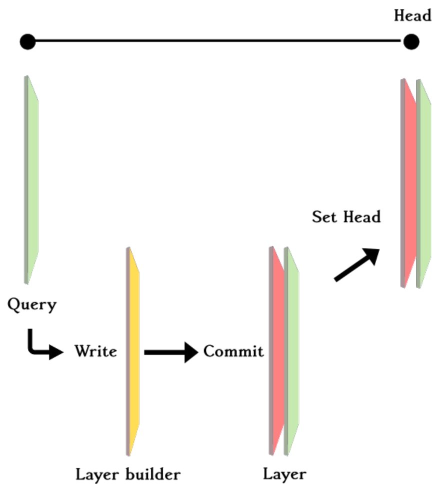

Figure 2: *Write transaction workflow*

the 20-byte identifiers. In the present implementation this is a file with the name of the label containing the 20-byte identifier.

2.3.1 Dictionary modfications

Due to the use of delta encodings, new triples can be added which are not present in the original dictionary. We therefore start new dictionaries with a recorded offset, remembering the last bucket from the previous dictionary.

2.4 Write Transactions

When an update transaction is initiated, a new *layer builder* is created, which logs all newly inserted or deleted edges. When this *layer builder* is committed, it yields a *layer* which has organised the insertions in our succinct data structures.

In TerminusDB we require that graphs conform to the constraints imposed by the OWL description of the dataset. This means that we produce a hypothetical database by committing the layer builder without advancing head. First we check the constraints hold on this new intermediate database and after these are passed, it is safe to advance head to this newly created layer. *Advancing* is done by side-effecting the label to point to the new 20-byte value. The problem of coordination in the face of side-effects is reduced to the problem of label management, simplifying much of the architecture. A schematic of the workflow of the write transaction is given in Figure 2.

2.5 Delta compression

As new updates are performed the database layer depth increases. This will incur a performance penalty requiring repeated searching at each layer for every query. In order to improve performance, we can perform a *delta compression* similar to the mechnaims used in git. Alternatively we can recalculate the full dataset as a new base-layer. In git this *delta compression* step can be performed manually or it will occur when a depth threshold is passed.

Since the layers are immutable, this operation can be done concurrently. Commis that occur before the process is completed simply layer over the delta with no visible change in the content of the database.

Compressed deltas of this type can allow older layers to be archived, or even deleted. The removal of previous layers removes the capacity to time-travel or to track whether the database arose from a branch. However, this information can be kept seperately in a metadata repository allowing memory of the branching structure and other information about previous commits, but not the capacity to time-tralve to them. We plan to implement this graph metadata repository in future version of TerminusDB.

3 Future Work

Values are stored as strings using a plain front coding dictionary uniformly for all data types. Obviously this is less than ideal in that it causes an expansion in size for the storage of integers, dates and other specific types. It also means that only search from the beginning of the datatype is optimised. In future versions of store we hope to differentiate our indexing strategies for the various datatypes in XSD.

For strings the use of succinct data structure immediately suggests a potential candidate: the FM-index[4]. With FM-indexing very large datasets could still have reasonable query times for queries which are typically done on full text indexes using inverted term-document indexing. We have yet to explore the candidates for numeric and date types.

Currently the tracking of history and branches is implicit. We intend to adopt a more explicit approach, storing a graph of the various commits coupled with timestamps and other metadata which will facilitate effective management.

4 Conclusion

The use of advanced CI/CD workflows for databases as yet has not been practical due to the lack of tool-chain support. In the software world we have seen just what a large impact appropriate tools can make with advent of git.

TerminusDB makes possible these collaborative CI/CD type operations in the universe of data management.

This is made possible because of the synergies which an immutable layered approach has with the *succinct* *datastructure* approach that we have used for encoding.

TerminusDB provides a practical tool for enabling branch, merge, rollback and the various automated and manual testing regimes which are faciliated by them on a transactional database management system which can provide sophisticated query support.

Bibliography

[1] Jim Blandy. *The Rust Programming Language: Fast, Safe, and Beautiful*. O'Reilly Media, Inc., 2015. ISBN: 9781491925447.

[2] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[3] Kevin Chekov Feeney, Gavin Mendel-Gleason, and Rob Brennan. "Linked data schemata: Fixing unsound foundations". In: *Semantic Web* 9.1 (2018), pp. 53–75. DOI: 10.3233/SW-170271. URL: https://doi.org/10.3233/SW-170271.

[4] Paolo Ferragina and Giovanni Manzini. "Indexing Compressed Text". In: *J. ACM* 52.4 (July 2005), pp. 552–581. ISSN: 0004-5411. DOI: 10.1145/1082036.1082039. URL: http://doi.acm.org/10.1145/1082036.1082039.

[5] Simon Gog and Matthias Petri. "Optimized succinct data structures for massive data". In: *Software: Practice and Experience* 44.11 (2014), pp. 1287–1314. DOI: 10.1002/spe.2198. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2198. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2198.

[6] Guy Joseph Jacobson. "Succinct Static Data Structures". AAI8918056. PhD thesis. Pittsburgh, PA, USA, 1988.

[7] G. E. Kaiser, D. E. Perry, and W. M. Schell. "Infuse: fusing integration test management with change management". In: *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*. 1989, pp. 552–558. DOI: 10.1109/CMPSAC.1989.65147.

[8] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. "Problems, causes and solutions when adopting continuous delivery—A systematic literature review". In: *Information and Software Technology* 82 (2017), pp. 55 –79. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.10.001. URL: http://www.sciencedirect.com/science/article/pii/S0950584916302324.

[9] Miguel A. Martínez-Prieto, Mario Arias Gallego, and Javier D. Fernández. "Exchange and Consumption of Huge RDF Data". In: *The Semantic Web: Research and Applications*. Ed. by Elena Simperl et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 437–452. ISBN: 978-3-642-30284-8.

[10] Miguel A. Martínez-Prieto et al. "Practical compressed string dictionaries". In: *Information Systems* 56 (2016), pp. 73 –108. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2015.08.008. URL: http://www.sciencedirect.com/science/article/pii/S0306437915001672.

[11] C. Mohan, Hamid Pirahesh, and Raymond Lorie. "Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-only Transactions". In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. SIGMOD '92. San Diego, California, USA: ACM, 1992, pp. 124–133. ISBN: 0-89791-521-6. DOI: 10.1145/130283.130306. URL: http://doi.acm.org/10.1145/130283.130306.

[12] Mohammad Sadoghi et al. "Reducing Database Locking Contention Through Multi-version Concurrency". In: *Proc. VLDB Endow.* 7.13 (Aug. 2014), pp. 1331–1342. ISSN: 2150-8097. DOI: 10.14778/2733004.2733006. URL: http://dx.doi.org/10.14778/2733004.2733006.