The Memory & Storage Design of Neo4J

Fabian Klopfer, Theodoros Chondrogiannis

December 3, 2020

Abstract

In this document I describe the internals of the storage layer of the popular native graph database Neo4J. A detailed description of how records are stored into which files and how these are accessed is to be elaborated on in the following article. This is a translation, update and extension of previous work done by Michael Brendle. Further the page cache (or buffer manager) and examples on access patterns generated by graph traversals are given. This is done in order to gain insights and prepare for optimizing locality on the storage level to minimize IO.

Contents

1 Introduction

1.1 Database architecture

Relational databases store tables of data. The links considered in this category of DBMS are mostly used to stitch together the fields of an entry into one row again, after it has been split to satisfy a certain normal form. Of course one may also store tables where one table stores nodes and the other table's fields are node IDs to represent relationships.

However, in order to traverse the graph, one has either to do a lot of rather expensive look ups or store auxiliary structures to speed up the look up process. In particular when using B-trees as index structure, each look up takes $\mathcal{O}(\log(n))$ steps to locate a specific edge. Alternatively one could store an additional table that holds edge lists such that the look up of outgoing or incomming edges is only $\mathcal{O}(\log(n))$ which would speed up breadth first traversals. But still one has to compute joins in order to continue the traversal in terms of depth. Another way to speed things up is to use a hash-based index, but this also has a certain overhead aside from the joins.

In contrast to relational data base management systems, native graph databases use structures specialised for this kind of queries. In the remainder of the document I discuss based upon Michael Brendle's work what structures and mechanisms the graph database Neo4J uses in order to achieve this superior performance in the domain of graphs.

First of all, let us consider the high level architecture of a database management systems as shown in figure 1.1 — with a focus on the storage and loading elements.

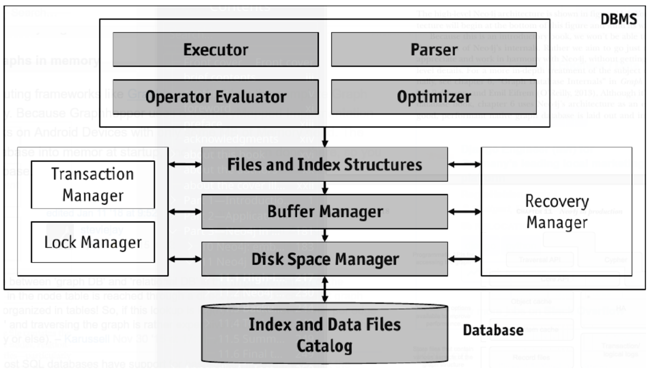


Figure 1: The typical structure of a relational database management system.

Here The disk space manager, sometimes also called storage manager, handles de-/allocations, reads & writes and provides the concept of a page: A disk block brought into memory. For that it needs to keep track of free blocks in the allocated file. Optimally both a disk block and a page are of the same size. One crucial task of a disk space manager is to store sequences of pages into continuous memory blocks in order to optimize data locality. Data locality has the upside, that one needs only one I/O operation to load multiple pages. To summarize the two most important objectives of a storage manager are to provide a locality-preserving mapping from pages to blocks based upon the information in the DBMS and to abstract physical storage to pages, taking care of allocation and access. A buffer manager is used to mediate between external storage and main memory. It maintains a designated pre-allocated area of main memory — called the buffer pool — to load, cache and evict pages into or from main memory. It's objective is to minimize the number of disk reads to be executed by caching, pre-fetching and the usage of suitable replacement policies. It also needs to take care of allocating a certain fraction of pages to each transaction.

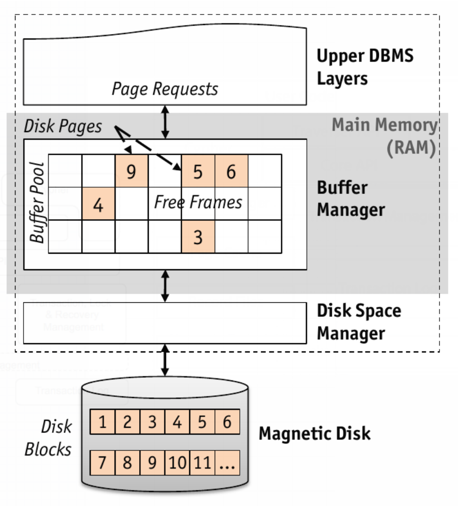


Figure 2: A visualization of the interaction of a database with memory.

The final memory and storage model relevant component of the of a database management system is the file layout and possible index structures. In order to store data a DBMS may either store one single or multiple files to maintain records.

A file may consist of a set of pages containing a set of slots. A slot stores one record with each record containing a set of fields. Further the database needs to keep track of free space in the file: A linked list or a directory must record free pages and some structure needs to keep track of the free slots either globally or per page.

Records may be of fixed or of variable size, depending on the types of their fields. Records can be layout in row or column major order. That is one can store sequences of tuples or sequences of fields. The former is beneficial if a lot of update, insert or delete operations are committed to the database, while the latter optimizes the performance when scans and aggregations are the most typical queries to the system.

Another option is to store the structure of the records along with pointers to the values of their fields in one files and the actual values in one or multiple separate files. Also distinct types of tables can be stored in different files. For example entities and relations can be stored in different files with references to each other, thus enabling the layout of these two to be specialized to their structure and usage in queries.

Files may either organize their records in random order (heap file), sorted or using a hash function on one or more fields. All of these approaches have upsides and downsides when it comes to scans, searches, insertions, deletions and updates.

To mitigate the effect that result from selecting one file organization or another, the concept of indexes have been introduced. Indexes are auxiliary structures to speed up certain operations that depend on one field. Indexes may be clustered or unclustered. An index over field F is called clustered if the underlying data file is sorted according to the values of F. An unclustered index over field G is one where the file is not sorted according to G. In a similar way indexes can be sparse or dense. A sparse index has less index entries than records, mostly one index entry per file. This can of course only be done for clustered indexes as the sorting of the data file keeps the elements between index keys in order. An index is dense if there is a one to one correspondence between records and index entries. There are different variants of storing index entries which have again certain implications on the compactness of the index and the underlying design decisions.

In another view of database management systems architectures, this boils down to the design decisions one has to make when implementing the storage layer and the access layer as shown in figure 1.1.

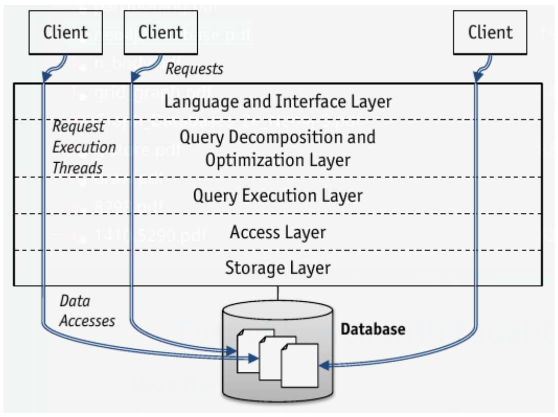


Figure 3: The architecture of a database management system from another point of view.

Here the storage layer is in close correspondence to the disk space manager in combination with the buffer manager, while the files and index structures provide the access layer.

All these considerations make choosing different file splits, layouts, orderings, addressing schemes, management structures, de-/allocation schemes and indexes a complex set of dependent choices. These depend mainly on the structure of the data to be stored and the queries to be run.

1.2 The Architecture of Neo4J

When restricting to graph structures where nodes and relationships are allowed to have properties and labels and types respectively, this allows one to narrow down some of the design decisions. In particular the example of a popular graph native database — Neo4J — is what I discuss in the next sections.

To get an overview of the architecture let us consider figure 1.2.

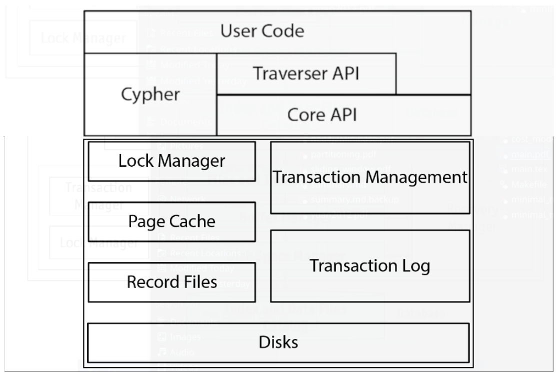


Figure 4: The high level architecture of Neo4J according to Emil Efrim, the co-founder of Neo technologies.

Here we can see that the previous schema is not exactly straight forward to apply, mainly due to a lack of concise documentation. The diagram was taken from the only publication that elaborates on the internals of Neo4J aside from the code of course. Here The storage manager makes mostly use of the Java NIO package with some additional usage of operating system native calls to allocate memory for the page cache and network buffers. A more detailed view on the high level architecture of the disk space and buffer manager and the fiels and index structures was deduced by the author from the source code and the non-public JavaDocs. This is shown in figure 1.2.

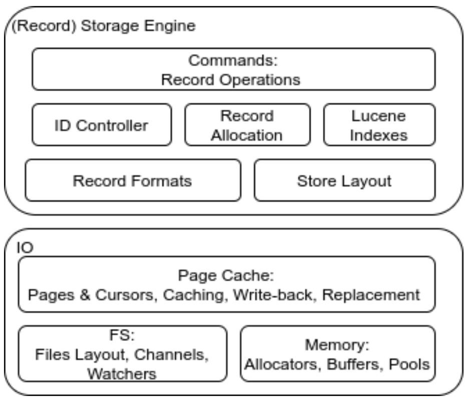


Figure 5: A visualization of the broad the storage and memory organization of Neo4J.

In the next section the focus is set on the storage layer of Neo4J: How it handles allocations, read and writes and how it provides the notion of a page. After that I elaborate on the details of the page cache, the transformation it applies to nodes and relationships on loading, the caching and the page eviction strategies it employs. Finally the internals of the files and records layouts are discussed along with the special structures employed and the indexes that are (or may be) created over the files.

The overall memory and storage state of a Neo4J instance and its environment may thus be visualized like this figure 1.2.

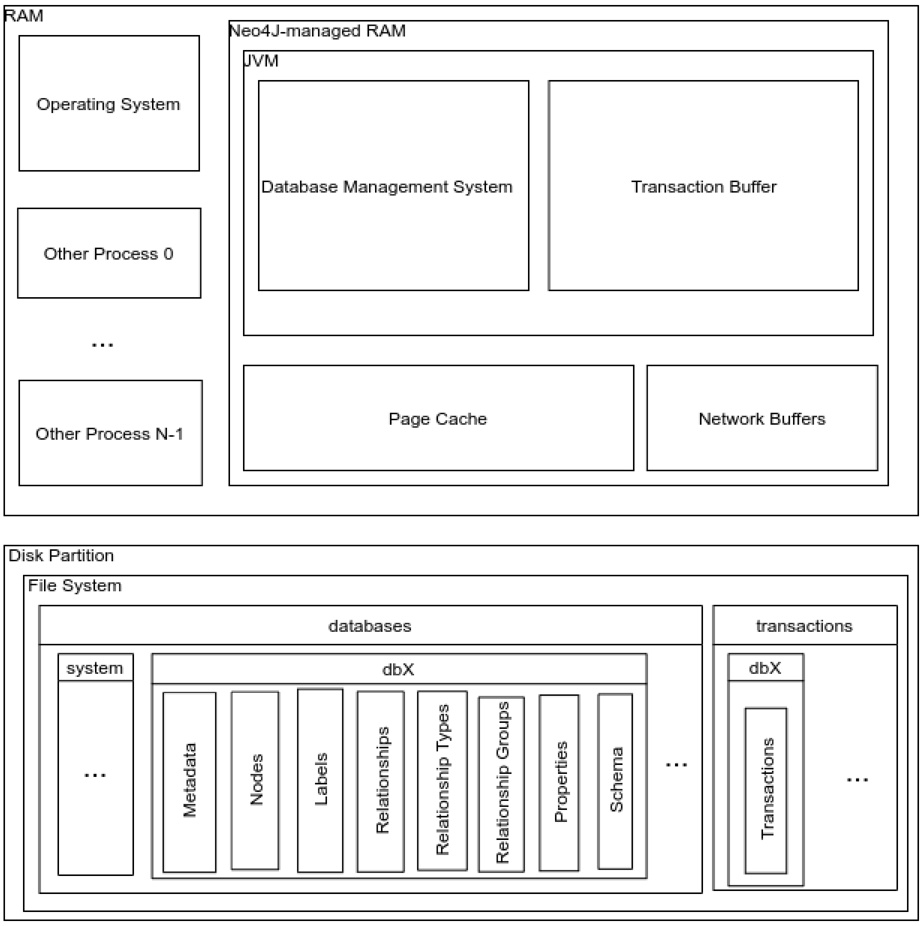


Figure 6: A sketch of how Neo4J occupies memory.

1.3 The Property Graph Model

TODO

2 Disk Space Management

TODO later
`neo4j/community/io/src/main/java/org/neo4j/io/(fs|mem|memory)`
`neo4j/community/native`
G-Store

3 Buffer Management

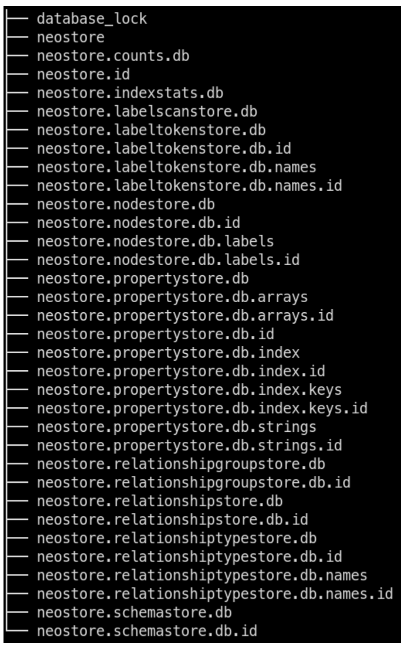
TODO later
`neo4j/community/io/src/main/java/org/neo4j/io/pagecache`
Page Cache record layout

4 File, Record & Index Structures

4.1 File Layout

The files that Neo4J uses to store data are categorizable into 5 classes based upon the records they contain:

- Node-related files: These files start with the prefixes `neostore.nodestore*` and `neostore.label*`. The contents of these files contain the node structure and labels.
- Relationship-related files: The prefix `neostore.relationship*` is used for all files containing records related to the graph's relationship structure, types and possibly groups.
- Property-related files: Containing the properties of both nodes and relationship, this is the part that is least structured and rather unspecialized with respect to graphs.
- Schema: Files starting with `neostore.schemastore*` contain information about the schema of the graph or rather the schematic information on the node, relationship and property labels, types and constraints.
- Metadata: Files only starting with `neostore` and none of the above prefixes contain meta information about the graph.
- Finally, a lock file.



```
├── database_lock
├── neostore
├── neostore.counts.db
├── neostore.id
├── neostore.indexstats.db
├── neostore.labelscanstore.db
├── neostore.labeltokenstore.db
├── neostore.labeltokenstore.db.id
├── neostore.labeltokenstore.db.names
├── neostore.labeltokenstore.db.names.id
├── neostore.nodestore.db
├── neostore.nodestore.db.id
├── neostore.nodestore.db.labels
├── neostore.nodestore.db.labels.id
├── neostore.propertystore.db
├── neostore.propertystore.db.arrays
├── neostore.propertystore.db.arrays.id
├── neostore.propertystore.db.id
├── neostore.propertystore.db.index
├── neostore.propertystore.db.index.id
├── neostore.propertystore.db.index.keys
├── neostore.propertystore.db.index.keys.id
├── neostore.propertystore.db.strings
├── neostore.propertystore.db.strings.id
├── neostore.relationshipgroupstore.db
├── neostore.relationshipgroupstore.db.id
├── neostore.relationshipstore.db
├── neostore.relationshipstore.db.id
├── neostore.relationshiptypestore.db
├── neostore.relationshiptypestore.db.id
├── neostore.relationshiptypestore.db.names
├── neostore.relationshiptypestore.db.names.id
├── neostore.schemastore.db
└── neostore.schemastore.db.id
```

Figure 7: A visualization of the files layout of Neo4j.

Most importantly the files ending with .db contain all fixed size records representing the structure of the graph and small amounts of data. Files ending with .id are used for record allocation and contain unused ids. Files having one of the suffixes names, arrays and strings are so called dynamic stores and contain dynamic sized records. Files ending with .index contain indexes either generated by Neo4J (e.g. the keys of properties are referenced like this to avoid storing key names multiple times) or by the Apache Lucene indexing library.

4.2 Record Structures

4.2.1 Address Translation

Neo4j defines the following constants as the size of the addresses used. These

```
public static final int PROPERTY_TOKEN_MAXIMUM_ID_BITS = 24;
        static final int NODE_MAXIMUM_ID_BITS = 35;
        static final int RELATIONSHIP_MAXIMUM_ID_BITS = 35;
        static final int PROPERTY_MAXIMUM_ID_BITS = 36;
        public static final int DYNAMIC_MAXIMUM_ID_BITS = 36;
        public static final int LABEL_TOKEN_MAXIMUM_ID_BITS = 32;
        public static final int RELATIONSHIP_TYPE_TOKEN_MAXIMUM_ID_BITS = 16;
        static final int RELATIONSHIP_GROUP_MAXIMUM_ID_BITS = 35;
        public static final int SCHEMA_RECORD_ID_BITS = 32;
```

are represented by the datatype long when loaded into main memory. On disk these values are stored as 32-bit Integers with additional modifiers, that is the highest bits that do not fit into an int are stored to fields with additional space.

For example the in-use bit consumes one byte of memory and the additional 7 bytes are used to stroe the highest bits that do not fit into an integer of another field.

Finally the base and the modifer are aggregated into a long by shifting both appropriately, casting them to longs and applying the logical or | operation.

Thus when refering to high bits and ID below, high bits do always mean the remainer of the bits that did not fit into the field that is said to be holding the ID. The latter is actually only holding the lowest 32 bits of an ID/address. The *actual ID* is always high bits and ID field, shifted appropriately and connected using the logical or operator.

4.2.2 Nodes

The record format of nodes consist of a 15 byte structure. The IDs of nodes are stored implicitly as their address. If a node has ID 100 we know that its record starts at offset 15 Bytes $\cdot$ 100 = 1500 from the beginning of the file. The struct of a record looks like this:

1. Byte 1: The first byte contains one bit for the in-use flag. The additional 7 bits are used to store the 3 highest bits of the relationship ID and the 4 highest bits of a property ID

2. Bytes 2 - 5: The next 4 Bytes represent the ID of the first relationship in the linked list containing the relationships of the considered node.

3. Bytes 6 - 9: Again 4 bytes encode the ID to the first property of the node.

4. Bytes 10 - 14: This 5 byte section points to the labels of this node, labels might also be inlined.
5. Byte 15: The last byte stores if the node is dense, i.e. one node has an aweful lot of relationships and is treated a bit differently. That is a relationships are stored by type and direction for this node into groups, see 4.2.6.

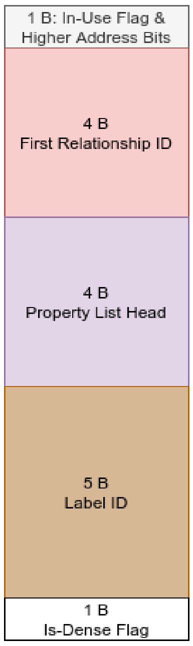

1 B: In-Use Flag & Higher Address Bits
4 B First Relationship ID
4 B Property List Head
5 B Label ID
1 B Is-Dense Flag

Figure 8: A visualization of the record structure of a node.

To summarize: The records on disk are stored as in the enumeration above. In the database all IDs get mapped to longs and their respective space is larger than the space representable by 35 bit — what is perfectly fine.

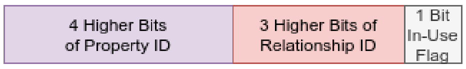

4 Higher Bits of Property ID
3 Higher Bits of Relationship ID
1 Bit In-Use Flag

Figure 9: A visualization of the information stored in the first byte of a node record.

On disk 4 byte integers are used to store the 32 lowest bits of the respective addresses and the higher bits are stored in the first byte that also carries the in-use bit.

4.2.3 Node Labels

Each node label record consists of an in use bit and a name ID, which in turn points to an entry in a seperate file storing label names as dynamic records (see

4.2.8), storing the label names. This is done in order to asure that the records are of fixed length.

1. Byte 1: In-use flag
2. Bytes 2-5: Pointer to the label string entry

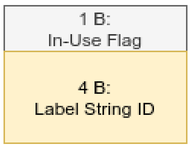


Figure 10: A visualization of the structure of a label record.

4.2.4 Relationships

Relationship records are stored with implicit IDs too. Their fixed size records contain 34 bytes. Besides an in-use flag and the node IDs that are connected, and the relationship type, the record also contains two doubly linked list: One for the relationships of the first node and one for the relationship of the second node. Finally a link to the head of the properties linked list of this relationship and a marker if this relationship is the first element in the relationships linked list of one of the nodes.

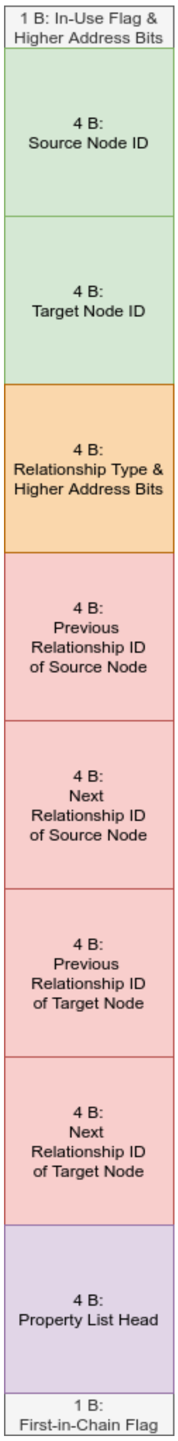

1 B: In-Use Flag & Higher Address Bits
4 B: Source Node ID
4 B: Target Node ID
4 B: Relationship Type & Higher Address Bits
4 B: Previous Relationship ID of Source Node
4 B: Next Relationship ID of Source Node
4 B: Previous Relationship ID of Target Node
4 B: Next Relationship ID of Target Node
4 B: Property List Head
1 B: First-in-Chain Flag

Figure 11: A visualization of the record structure of a relationship in Neo4J.

1. Byte 1: In-use bit, first node high order bits (3 bits), first property high order bits (4 bits)
2. Bytes 2 - 5: first node ID
3. Bytes 6 - 9: second node ID
4. Bytes 10 - 13: relationship type (16 bit), second node high order bits (3 bits), relationship previous and next ID higher bits for first and second node ($4 \cdot 3 = 12$ bits, one unused bit.
5. Bytes 14 - 17: previous relationship ID for first node
6. Bytes 18 - 21: next relationship ID for first node
7. Bytes 22 - 25: previous relationship ID for second node
8. Bytes 26 - 29: next relationship ID for second node
9. Bytes 30 - 33: link to the first property of the relationship
10. Bytes 34: A marker if this relation is the first element in the relationship linked list of one of the nodes stored in the lowest two bits of the byte. The other 6 bits are unused.

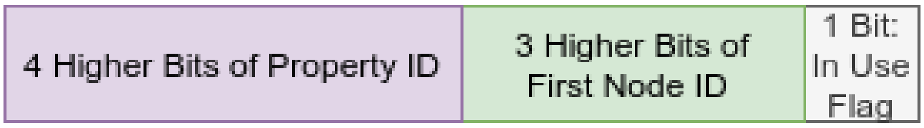


Figure 12: A visualization of how information is stored in the first byte of a relationship record.

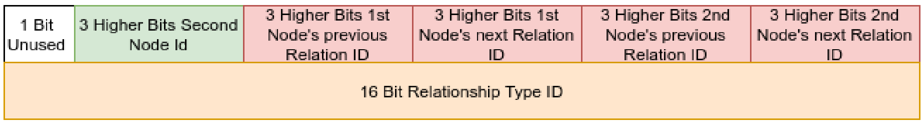

1 Bit Unused
3 Higher Bits Second Node Id
3 Higher Bits 1st Node's previous Relation ID
3 Higher Bits 1st Node's next Relation ID
3 Higher Bits 2nd Node's previous Relation ID
3 Higher Bits 2nd Node's next Relation ID
16 Bit Relationship Type ID

Figure 13: The structure of the bytes that are used to store the type of a relationship and high bits of a node and a relationship IDs.

4.2.5 Relationship Types

Similarly to the node labels, the relationship type records posses an in-use flag and a type ID that points to a record in a file containing strings in the dynamic record format.

1. Byte 1: In-use flag
2. Bytes 2-5: Pointer to the type string entry

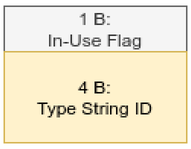


Figure 14: A visualization of the record structure of a relationship type.

4.2.6 Relationship Groups

The relationship group record is used for dense nodes in order not to iterate over all relationships but only over those of a specific type. Each record consists of 25 bytes where the first byte again contains the in-use flags and the high order bits of IDs that do not fit into integers. The next bytes again contains high bits of addresses and is followed by the relationship type. After that a reference (an ID) to the next relationship group record is given. Finally the first outgoing relation, the first incoming relation, the first looping relation and the node of interest are specified.

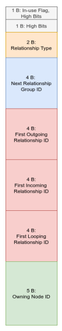

1 B: In-use Flag, High Bits
1 B: High Bits
2 B: Relationship Type
4 B: Next Relationship Group ID
4 B: First Outgoing Relationship ID
4 B: First Incoming Relationship ID
4 B: First Looping Relationship ID
5 B: Owning Node ID

Figure 15: A visualization of the record structure of a relationship group.

1. Byte 1: In-use flag, high bits for the next relationship group ID, high bits for the first entry of outgoing relationship list.
2. Byte 2: High bits for the first entry in the incoming relationship list, high bits for the first entry in the looping relationship list.
3. Bytes 3 and 4: Relationship type ID
4. Bytes 5 - 8: ID of the next relationship group record.
5. Bytes 9 - 12: ID of the first relationship in the linked list of outgoing relationships, i.e. those relationships which start at the record owning node.
6. Bytes 13 - 16: ID of the first relationship in the linked list of incoming relationships, i.e. those relationships which end at the record owning node.
7. Bytes 17 - 20: ID of the first relationship in the linked list of loops, i.e. relationships which start and end at the record owning node.

8. Bytes 21 - 25: ID of the node for which the relationship group was created, also called the owning node (in the source code).

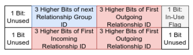

1 Bit: Unused
3 Higher Bits of next Relationship Group ID
3 Higher Bits of First Outgoing Relationship ID
1 Bit: In-Use Flag
1 Bit Unused
3 Higher Bits of First Incoming Relationship ID
3 Higher Bits of First Outgoing Relationship ID
1 Bit Unused

Figure 16: The structure of the first two bytes of a relationship group record.

4.2.7 Properties

Properties are stored per node or relationship as doubly linked list of 41 byte records with the first 9 bytes storing the previous and next property IDs and the remaining 32 bytes carry so called property blocks.

1. Byte 1: High bits of next and previous property IDs
2. Bytes 2 - 5: Previous property ID
3. Bytes 6 - 9: Next property ID
4. Bytes 10 - 41: Up to 4 Property Blocks

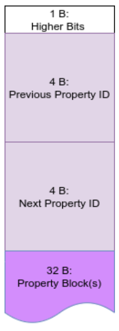


Figure 17: The structure of a property record.

Property Blocks Each propety block consists of one to four 8 byte value blocks. Thus each property block spans between 8 and 32 byte. The first value block of a property block has the following structure:

1. Byte 1 - 3: Property key index ID (pointer to the key name)
2. Byte 4: Type, inlined flag and 3 bits data
3. Byte 5 - 8: Data

All remaining value blocks contain pure data. Property values of type boolean, byte, short, int, char and float fit into one property block with one value block. Properties of type long and double are stored into two value blocks.

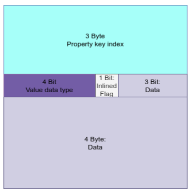


Figure 18: The structure of a property block.

Short strings and arrays are stored in the same property block as well, if they fit into 28 byte. Otherwise a reference is stored, that refers to one of the respective dynamic stores for array and string property values. Other types such as temporal and spatial data are also fit into one property block. Details on each data type are outlined in the comment from the source code shown below. As there are only 14 different types, 4 bits suffice to encode the type. The types are enumerated in the order given below starting at `0b0001`.

```
/*  1: BOOL
   *  2: BYTE
   *  3: SHORT
   *  4: CHAR
   *  5: INT
   *  6: LONG
   *  7: FLOAT
   *  8: DOUBLE
   *  9: STRING REFERENCE
   * 10: ARRAY  REFERENCE
   * 11: SHORT STRING
   * 12: SHORT ARRAY
   * 13: GEOMETRY
   */
```

Figure 19: The type enumeration used for property blocks

Property Key Index The property key index contains a in-use flag, a usage count and a reference to the respective name string in a dynamic store.

1. Byte 1: In-use Flag
2. Byte 2 - 5:

4.2.8 Dynamic Records: Strings & Arrays

A record in a dynamic storeis either strings or arrays stored as properties, the strings of node labels or relationsgip types each of which stored in their own dynamic store. A dynamic record has 8 fixed bytes followed by up to $2^2 4$ bytes = 16MiB. If the string or array to be stored is larger, then the remainder of the data is stored using a linked list of next blocks. The absolute amount of

```
/* BOOL:      [    ,    ] [    ,    ] [    ,    ] [    ,    ] [   x,type][K][K][K]
    * BYTE:      [    ,    ] [    ,    ] [    ,    ] [    ,xxxx] [xxxx,type][K][K][K]
    * SHORT:     [    ,    ] [    ,    ] [    ,xxxx] [xxxx,xxxx] [xxxx,type][K][K][K]
    * CHAR:      [    ,    ] [    ,    ] [    ,xxxx] [xxxx,xxxx] [xxxx,type][K][K][K]
    * INT:       [    ,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,type][K][K][K]
    * LONG:      [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxx1,type][K][K][K] inlined
    * LONG:      [    ,    ] [    ,    ] [    ,    ] [    ,    ] [   0,type][K][K][K] value in next block
    * FLOAT:     [    ,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,type][K][K][K]
    * DOUBLE:    [    ,    ] [    ,    ] [    ,    ] [    ,    ] [    ,type][K][K][K] value in next block
    * REFERENCE: [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,type][K][K][K]
    * SHORT STR: [    ,    ] [    ,    ] [    ,    ] [    ,   x] [xxxx,type][K][K][K] encoding
    *            [    ,    ] [    ,    ] [    ,    ] [ xxx,xxx ] [    ,type][K][K][K] length
    *            [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [x   ,    ] payload(+ maybe in next block)
    *                                                           bits are densely packed, bytes torn across blocks
    * SHORT ARR: [    ,    ] [    ,    ] [    ,    ] [    ,    ] [xxxx,type][K][K][K] data type
    *            [    ,    ] [    ,    ] [    ,    ] [  xx,xxxx] [    ,type][K][K][K] length
    *            [    ,    ] [    ,    ] [    ,xxxx] [xx  ,    ] [    ,type][K][K][K] bits/item
    *                                                                                 0 means 64, other values "normal"
    *            [xxxx,xxxx] [xxxx,xxxx] [xxxx,    ] [    ,    ] payload(+ maybe in next block)
    *                                                           bits are densely packed, bytes torn across blocks
    * POINT:     [    ,    ] [    ,    ] [    ,    ] [    ,    ] [xxxx,type][K][K][K] geometry subtype
    *            [    ,    ] [    ,    ] [    ,    ] [    ,xxxx] [    ,type][K][K][K] dimension
    *            [    ,    ] [    ,    ] [    ,    ] [xxxx,    ] [    ,type][K][K][K] CRSTable
    *            [    ,    ] [xxxx,xxxx] [xxxx,xxxx] [    ,    ] [    ,type][K][K][K] CRS code
    *            [    ,   x] [    ,    ] [    ,    ] [    ,    ] [    ,type][K][K][K] Precision flag: 0=double, 1=float
    *            values in next dimension blocks
    * DATE:      [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxx1] [ 01 ,type][K][K][K] epochDay
    * DATE:      [    ,    ] [    ,    ] [    ,    ] [    ,   0] [ 01 ,type][K][K][K] epochDay in next block
    * LOCALTIME: [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxx1] [ 02 ,type][K][K][K] nanoOfDay
    * LOCALTIME: [    ,    ] [    ,    ] [    ,    ] [    ,   0] [ 02 ,type][K][K][K] nanoOfDay in next block
    * LOCALDTIME:[xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [ 03 ,type][K][K][K] nanoOfSecond
    *            epochSecond in next block
    * TIME:      [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [ 04 ,type][K][K][K] secondOffset (=ZoneOffset)
    *            nanoOfDay in next block
    * DATETIME:  [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxx1] [ 05 ,type][K][K][K] nanoOfSecond
    *            epochSecond in next block
    *            secondOffset in next block
    * DATETIME:  [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxx0] [ 05 ,type][K][K][K] nanoOfSecond
    *            epochSecond in next block
    *            timeZone number in next block
    * DURATION:  [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [xxxx,xxxx] [ 06 ,type][K][K][K] nanoOfSecond
    *            months in next block
    *            days in next block
    *            seconds in next block
    */
```

Figure 20: The layout of different types in property blocks.

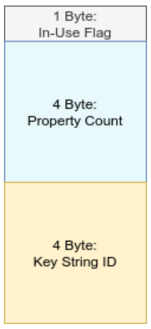


Figure 21: The structure of a property key index record.

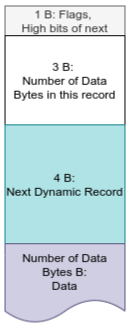


Figure 22: The structure of a dynamic record.

dynamic data that can be stored that way is only limited by the machines disk space, so potentially infinite (as the admin could keep on buying new disks). The structure of such a record is as follows:

1. 1 Byte: Start or linked record flag, in-use flag, high bits of next block ID
2. 2 - 4: Number of bytes used to store data beginning after the header.
3. 5 - 8: Reference to the next dynamic record
4. 9 - Number of bytes used - 1: Data

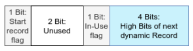


Figure 23: The first byte of a dynamic record.

4.2.9 Schema

TODO later

4.2.10 Metadata

TODO later

4.3 Record Allocation

TODO later
Reusing space
How delete works

4.4 Indexes

TODO later

5 Examples

First we give an example graph in the visual form: A simple social network that is created using the following Cypher query:

```
CREATE (Bob:User {id : 1,name:'Bob'})
CREATE (Peter:User {id : 2,name:'Peter'})
CREATE (Anna:User {id : 3,name:'Anna'})
CREATE (Amy:User {id : 4,name:'Amy'})
CREATE
    (Bob)-[:knows]->(Peter),
    (Bob)-[:knows]->(Amy),
    (Anna)-[:knows]->(Peter),
    (Anna)-[:knows]->(Amy),
    (Peter)-[:knows]->(Amy)
```

5.1 Purely disk-based Access

5.1.1 BFS Traversal

5.1.2 Dijkstra Traversal

5.1.3 A*-Traversal

5.2 Caching-assisted Access

5.2.1 BFS Traversal

5.2.2 Dijkstra Traversal

5.2.3 A*-Traversal

6 Conclusion