ABSTRACT
There is an emerging interest in graph databases due to their potential in storing and querying ubiquitous and ever-growing connected data. Cypher, the most widely deployed graph query language, is simple to learn and allows for compact expression of graph queries. However, given the intrinsic complexity of many graph algorithms and the expansive horizon of possible optimizations, the subject of query processing over large graphs, including how to efficiently represent and store a graph’s structure and data to support querying, can appear daunting. In this paper, we propose a graph representation based on adjacency lists that keeps the graph structure and node properties entirely in Redis, a popular in-memory key-value store. Next, we outline a smaller and less expressive graph query language, Cypher--, that “dumbs down” the original by removing advanced features, such as the Kleene star and GREP-like path expressions, while remaining a proper subset of the language. Armed with the above, as well as several assumptions about the graph at hand, we propose a simple query evaluation algorithm that relies on associating variables to the set of their possible bindings and leverages Redis’s $O(1)$ look up time. The correctness of this algorithm is assessed by comparing the output of our prototype against Neo4j, which is treated as the reference implementation. Finally, we compare the performance our implementation with Neo4j’s and describe a class of queries where our approach produces the same results and outperforms Neo4j by up to an order of magnitude, even with indices enabled.

Keywords
Neo4j, Cypher, Redis, graph query processing

1. INTRODUCTION
The data model inherent in the Neo4j and other popular graph databases is the labeled property graph [2]. This model treats the graph as a set of labeled nodes connected with named and directed edges, with both serving as containers to hold properties [12]. Unlike traditional relational databases, Neo4j exhibits a property called index-free adjacency, which entails that each vertex hold a pointer to its neighboring vertices, allowing graph walking to be a purely local operation which doesn’t need to consult a global index. This allows, to some extent, for query performance to be constant to the size of the dataset, assuming that the density does not also increase with node count. Neo4j achieves this by separating the storage of graph structure from that of property data, and using fixed-size records that allow the offset of any individual record to be computed in $O(1)$ time, given its ID.

In our implementation, we aim to achieve similar $O(1)$ performance by storing the graph in Redis, a popular and performant in-memory key-value store that allows simple data structures (lists, sets, hash maps, and sorted sets) or flat values to be associated with a string key and be retrieved in constant time. Although Redis configured by default to incrementally flush its state to nonvolatile memory, its performance depends on operating on the BASE model of Basic Availability, Soft-State, and Eventual Consistence.

In the sections that follow, we will explain our methodology for storing a labeled property graph in Redis, describe the Cypher– query language, and examine the proposed query evaluation algorithm in more detail.
that label. Remember that a node may have multiple labels.

4. For every unique mapping of property to value among the properties associated with the graph’s nodes, insert key index:property:value associated with the set of nodes such that for every node in the set, its property field is set to value.

5. Keep the following bookkeeping (key,value) pairs:
   - NODES associated with the set of all node IDs
   - LABELS associated with the set of all node labels
   - RTYPES associated with the set of all edge types

Item 4 specifies keeping an index mapping every property value to its corresponding set of nodes. This indexing adds redundancy to the representation, and its storage cost is explored further in Section 6.3. Additionally, to allow for long property values (length 255 or more), which are not advisable to use as Redis keys, we hash these values to a 16-byte md5 hash and use that in index keys. We also keep a set of properties for which we’ve chosen to do this. To prevent returning an incorrect output in the case of hash collisions, one can optionally check the full property value as a final query postprocessing step.

A concrete example of the graph representation for a small network of connected products is shown in Figure 1.

### 3. THE CYPHER-- LANGUAGE

In Neo4j, the property graph we have described can be queried by way of a native Java API, or a declarative query language called Cypher, which invites comparisons to SQL despite having significant structural and syntactic differences. Cypher allows for both query and DDL/DML language, though our Cypher-- dialect is a subset strictly of Cypher’s query features. In particular, it supports the three most essential Cypher clauses: MATCH, WHERE, and RETURN.

To introduce the main limitation imposed by Cypher--, we begin by considering a typical Cypher query, and it’s Cypher-- counterpart. This is also a prototypical example of the kind of query we build our prototype to handle:

```
MATCH (a:Person)-[:KNOWS]->(b)-[:ACTS]->(c:Movie)
WHERE a.age=25 AND (c.title="Forrest Gump" or c.title="Titanic")
RETURN a
```

The above finds all people age 25 who know someone who has starred in either the movie Forrest Gump or Titanic. In Cypher--, would be written as:

```
MATCH (a)-[r1]->(b), (b)-[r2]->(c)
WHERE a:Person AND c:Movie AND type(r1)="KNOWS" AND type(r2)="ACTS" AND a.age=25 AND (c.title="Forrest Gump" or c.title="Titanic")
RETURN a
```

We highlight the following syntactic differences:

- All property, label, and edge type conditions have been stripped from the MATCH clause, allowing it to specify only the path structure.
- Paths of length 3 or more are required to be expressed as multiple comma-separated parts, enabling them to be evaluated in isolation.
- All edges in the MATCH must either be bidirectional or point forward.

To parse Cypher-- queries, we use the ANTLR4 [11][10] parser generator library, that takes as input a grammar and outputs a parser that builds and walks a parse tree. An ANTLR4 grammar, which resembles an Context-Free Grammar in Extended Backus-Naur Form, can be found in Appendix A.
4. QUERY PROCESSING
At a high level, our query processing algorithm starts by visiting the WHERE clause and evaluating it recursively to arrive at a set of contexts, where each context consists of an assignment to every variable of its set of possible node ID’s or edge types. We call this set the context universe. Then, the algorithm visits every sub-pattern in the MATCH clause and finds the possible bindings of nodes to variables in that sub-pattern, for every context in the context universe. The results for every context are then joined to form the sub-pattern result. Next, the sub-pattern results for all sub-patterns are joined on the common variables. Finally, the relevant fields of every node in the resultant table are output to the user.

4.1 Where Clause
The WHERE clause is evaluated recursively, forming a tree whose leaves are type, label, or property conditions. Each vertex in this tree represents a set of contexts, where a context is an assignment of possible node bindings to every variable. The root represents the context universe, whereas the leaves are singleton sets consisting of one context where exactly one variable is constrained, and all other variables are unconstrained. Context sets are combined with Boolean operators, as is explained below and illustrated in Figure 2, where the ellipses represent contexts and the dashed rectangles represent context sets.

4.1.1 Boolean Operators: AND
The AND operator is the simplest of the three and is implemented by pairwise conjunction between all pairs in the left and right child’s context set. This is also the only Boolean operator currently supported by our prototype.

4.1.2 OR
The OR operator takes two context sets and unions them into a context set which contains the union of its two children. Since this results in duplicates (tuples in both of the contexts), a post-processing step will be needed to remove them.

4.1.3 NOT
The NOT operator is implemented by applying De Morgan’s Laws in order to transform the context set (which acts as a disjunction or union of multiple contexts) into a conjunction where the NOT operator is applied to each individual context. In order to negate a single context (which is conjunctive in itself), we again apply De Morgan’s Laws. This yields a disjunction of n contexts, one for every variable x which was restricted in the context being negated, where the result context has variable x set to the complement of its previous value and all other variables are unrestricted.

4.2 Match Clause
The next step is to produce tables of variable bindings for each sub-pattern in the MATCH clause, and then inner join these tables on the common variables. To accomplish this, we visit each sub-pattern and evaluate it for every context in the context universe, appending the output of each pass to the overall result for the sub-pattern. When building a binding for binding table for a sub-pattern, there are two cases to consider: either the pattern is a single node, or an edge between two nodes. If it is a single node, the output table is identical to the variable’s context. Otherwise, iterate over the nodes for the variable which has the smaller context. For each node, retrieve its neighborhood as defined by the permissible edge types (if there is an edge variable in the sub-pattern which restricts the edge type) and the edge direction. For every node in the neighborhood which is also in the context of the other variable in the sub-pattern, output the pair to the binding table.

Doing an inner join between sub-pattern results is accomplished by hashing the results for one of the sub-patterns and iterating over the results of the other sub-pattern.

4.3 Return Clause
The output of visiting the match clause is a table of results whose columns are the variables in the match clause. To output the desired properties to the user, we iterate through the rows of the result table and perform another Redis lookup for every node bound to a variable the user has specified should be output. From the returned hash map, we select the properties specified.

4.4 Pitfalls and Idiosyncrasies

Two implementation difficulties were encountered in the developing our prototype that are notable enough to deserve further attention, as they provide some insight into the Cypher semantics. The first is self loops. The second is Cypher’s choice to only follow a given edge once in the entirety of the pattern evaluation.

4.4.1 Self Loops

Self loops are fully permitted in the property graph data model, and are supported by Neo4j. However, given our more strict graph structure, we restrict our graph to have no more self loops than the number of edge types, as an edge must be unique identifiable by its source, target, and type. Self loops require special attention because they are easy to double count. Consider the following query: MATCH (a)-(b) RETURN a, b. Here, the context of a and b is set to NODES, the IDs of all vertices in the graph. In evaluating the pattern results, the algorithm will iterate through each a in NODES and find all its in and out neighbors which are also in NODES. If vertex v has a self loop, causing v to be both an in neighbor and out neighbor of itself, the Cypher semantics specifies that only one of these pairings should be added for the final output.

We can also note that one can explicitly asks for all self loops with a pattern such as MATCH (a)-(a) RETURN a, which our prototype supports.

4.4.2 Revisiting Relationships

Consider a graph with only two vertices, labeled as Mike and Amy. Say there is one edge of type "FriendsWith" from Mike to Amy, and another of the same type in the opposite direction. Now consider a query asking for nodes a,b,c where a is friends with b and b is friends with c run over our graph. Would Mike and Amy appear in the results? According to Cypher, they would not. But not because the traversal would revisit either Mike or Amy. Rather, it is because it would re-walk one of the two edges. This is Cypher’s solution to the problem of cycles of length two or more. As such, when computing sub-pattern results, this necessitates remembering which edge was taken between a node pair. Additionally, when joining sub-pattern results, one must check that the output rows don’t contain duplicate edges, adding further complexity to the join step.

5. RESULTS AND DISCUSSION

In this section, we compare the performance of KVGraph to Neo4j, both indexed and not indexed. As was previously mentioned, the present prototype is built to handle only conjunctive queries, so that is our testing ground. We perform our testing on a personal workstation.

5.1 Dataset

The graph dataset we have chosen for testing is of product metadata from Amazon, spanning May 1996 - July 2014 [8][7]. Since the entirely of this data would overwhelm main memory, we chose a subset of Electronics items. This data was contained 498,196 items, each of which had numerous properties from which we chose the item’s asin (unique Amazon identifier), title, price, brand, and sales rank. Each item can be connected to an arbitrary number of other items through three relationship types: "ALSO VIEWED" (AV), "ALSO BOUGHT" (AB), and "BOUGHT TOGETHER" (BT). After extracting the properties we wanted, the result was four CSV files (one with the items, and the other three holding source,target relationship tuples for each of the three relationship types) totalling 590 Megabytes. The graph had 9.56 million edges.

5.2 Experimental Results

Table 1 presents the running times of five selected conjunctive queries. We see that KVGraph easily beats Neo4j on all queries where variable contexts are reasonably restricted. However, where it falls short are queries where a large number of variables have unrestricted contexts. Since sub-pattern results are computed independently and then joined, KVGraph does not walk a path in the same way Neo4j does, which can be detrimental to performance.

<table>
<thead>
<tr>
<th>Query</th>
<th>Neo4j (no indices)</th>
<th>Neo4j (with indices)</th>
<th>KVGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>M (a) -&gt; (b) W a.asin='B0072M4R30' R a</td>
<td>2.41s</td>
<td>1.12s</td>
<td>4.73ms</td>
</tr>
<tr>
<td>M (a) -&gt; (b) W a.price='100.0' R b</td>
<td>1.70s</td>
<td>911ms</td>
<td>79.4ms</td>
</tr>
<tr>
<td>MATCH (a)-[r1]-(b),-[r2]-(c),-[r3]-(a) W a.brand=&quot;Panasonic&quot; AND b.brand=&quot;Canon&quot; AND c.brand=&quot;Nikon&quot; AND type(r1)=&quot;AV&quot; AND type(r2)=&quot;AV&quot; AND type(r3)=&quot;AV&quot;</td>
<td>2.84s</td>
<td>2.35s</td>
<td>622ms</td>
</tr>
<tr>
<td>MATCH (a) R a</td>
<td>4.19s</td>
<td>3.53s</td>
<td>102s</td>
</tr>
</tbody>
</table>

Table 1: Comparing performance of indexed and unindexed Neo4j with KVGraph for a selection of queries
6. CONCLUSION

In this paper, we presented an approach for keeping the structure and data of a graph database entirely in Redis, a key-value store. To test the feasibility of this approach with respect to allowing fast querying using the Cypher graph query language, we first defined a pared down version of Cypher, which, while simple, keeps true to Cypher’s basic principles. We then outlined an approach for query evaluation, and tested it on real world graph data. Our findings are encouraging, suggesting that Redis is a realistic medium for storing graph databases. However, more work needs to be done for queries where many individual variables are left unconstrained.

Appendix A

grammar CypherMM;
query : matchClause whereClause? returnClause ;
matchClause : 'MATCH 'pattern (',' pattern)*;
pattern : nodePattern patternElementChain*;
nodePattern : '('variable? ')';
patternElementChain : relationshipPattern nodePattern;
relationshipPattern : ( '-' relationshipDetail? '-' )
| ( '-' relationshipDetail? '->' )
relationshipDetail : '['variable? ']';
whereClause : 'WHERE' condition;
condition : '($('<condition ')
| 'NOT' condition
| condition 'AND' condition
| condition 'OR' condition
| labelCondition
| typeCondition
| propertyCondition
;
labelCondition : variable ':=' labelName (':=' labelName)*;
typeCondition : 'type' '('variable ')' '=' typeName ;
propertyCondition : variable '.' fieldName '=' fieldValue ;
returnClause : 'RETURN' (variable | (variable '.' fieldName)) (',' (variable | (variable '.' fieldName)))*;
variable : SymbolicName ;
labelName : SymbolicName ;
typeName : STRING ;
fieldName : SymbolicName ;
fieldValue : STRING ;
SymbolicName : [a-zA-Z] [a-zA-Z0-9]* ;
STRING : "" (ESC | ["\"\bfnrt] | UNICODE) ;
fragment ESC : '\\' (["\"] | UNICODE) ;
fragment UNICODE : 'u' HEX HEX HEX HEX ;
fragment HEX : [0-9a-fA-F] ;
WS : WHITESPACE+ -> skip;
WHITESPACE : ('' | '	' | '
' | '')+;

7. REFERENCES
[12] I. Robinson, J. Webber, and E. Eifrem. Graph

<table>
<thead>
<tr>
<th>CSV files</th>
<th>Neo4j</th>
<th>KVGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.59Gb</td>
<td>1.49Gb</td>
<td>2.23Gb</td>
</tr>
</tbody>
</table>

Table 2: Data size
Databases: New Opportunities for Connected Data. "  
O’Reilly Media, Inc.", 2015.
