Optimization of Graph Query on Relational Database

Nirav P. Chavda
Robinson College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

(Research Project - Option B)

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UNITED KINGDOM

Email: nkpc2@cl.cam.ac.uk

June 14, 2012
Declaration

I Nirav P. Chavda of Robinson College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,235

Signed:

Date:

This dissertation is copyright ©2012 Nirav P. Chavda.
All trademarks used in this dissertation are hereby acknowledged.
Acknowledgements

I would like to thank my supervisors Dr. Eiko Yoneki and Dr. Amitabha Roy for their valuable time, many suggestions, inspiring feedback and regular meetings regarding the research project. I highly appreciate the opportunity that was provided me to work on this project which combines novelty with practicality.

I would also like to thank my co-advisor Karthik Nilakant. He provided me with exceptional help and tremendous support throughout the lifetime of this research project. He dedicated countless hours of his private time to support and I am extremely grateful for that.
Abstract

Graph data structures are being increasingly employed to model dynamic information such as online social networks, transportation networks and protein structures. They are widely used due to the fact that they can represent real world knowledge through an intuitive and flexible scheme. These data structures allow intuitive interconnection between different nodes or components through edges. There can be more than one edge between a pair of nodes and each edge have a label which defines the relationship between connected nodes. Furthermore, graphs can easily capture real world dynamics through addition or removal of edges.

With the increase in popularity of graphs, various specialised graph database have arised such as Neo4j and Trinity. These NoSQL databases are designed so that they can be easily scaled and have high availability. However, the legacy Relational Database Management Systems (RDBMS) have key advantages. They have been employed by various industries for long time and its features have evolved through extensive research that has been carried out on the relational model. It includes features such as transaction consistency, normalization and declarative query language. Furthermore, there are various RDBMS off-the-shelf products available which can be quickly deployed. Due to various desirable properties, the RDBMS continue to be used for different applications. However, the relational database are not efficient on querying graphs as it involves JOIN operation between tuples which is slow and has limitations.

This project aims to address this issue by applying efficient pre-fetching techniques to optimize graph query. A graph processing layer called Crackle was also built which achieves local caching at the application level. A set of experiments are conducted to evaluate the performance gain through Hierarchical blocking technique. There was significant improvement in Small World by upto 8x followed by Random and Scale free graphs. The comparison tests between relational and graph database showed decrease in the performance gap between them. In some cases, the relational database outperformed the graph database by 18% by using Hierarchical blocking. Clearly, this shows how Hierarchical blocking can achieve implicit pre-fetching through efficient graph layout. This algorithm can be further extended by applying more targeted heuristic based on the topology of underlying graph. However, that would make the blocking algorithm more graph-specific as opposed to being domain-indepdent which is what this project aims to explore.
# Contents

1 Introduction .................................. 1
   1.1 Aims ........................................ 3

2 Background and Related Works ................. 4
   2.1 Relational Versus Graph Database Model .......... 4
   2.2 Graph representation .......................... 6
   2.3 Sources of latency ............................ 7
      2.3.1 Between database and application .......... 7
      2.3.2 Between disk and database ................ 8
   2.4 Reducing latency .............................. 9
      2.4.1 Caching at Crackle level .................. 9
      2.4.2 Efficient prefetching ....................... 10
   2.5 Prefetching and semantic locality .............. 11

3 Hierarchical Blocking .......................... 13
   3.1 Hot cache analysis of blocking effect .......... 21

4 System Overview ................................ 23
   4.1 Caching ...................................... 24

5 Data Preperation ................................ 27
   5.1 Graph pre-processing .......................... 28
      5.1.1 H-blocking ............................... 29
      5.1.2 Randomization ............................ 31

6 Evaluation ...................................... 34
   6.1 Batch 1 ....................................... 34
      6.1.1 Breadth First Search ....................... 35
      6.1.2 Depth First Search ......................... 37
      6.1.3 Single Source Shortest Path ............... 39
   6.2 Batch 2 ....................................... 41
6.2.1 Breadth First Search . . . . . . . . . . . . . . . . . . . . 42
6.2.2 Depth First Search . . . . . . . . . . . . . . . . . . . . 43
6.2.3 Single Source Shortest Path . . . . . . . . . . . . . . . 44
6.3 Comparison with Neo4j . . . . . . . . . . . . . . . . . . . . 45
6.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
   6.4.1 Effect of large node on H-block layout . . . . . . . . . 45
   6.4.2 H-block efficiency based on nature of query . . . . . . . 48
   6.4.3 Distance Effect . . . . . . . . . . . . . . . . . . . . . . 50

7 Future Works . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

8 Summary and Conclusions . . . . . . . . . . . . . . . . . . . . 55
# List of Figures

2.1 Block prefetching on road network .................................. 10

3.1 Memory hierarchy of a typical computer system with associated access cycles for each level ................................. 14
3.2 van Emde Boas layout scheme ........................................... 15
3.3 Undirected graph and memory hierarchy inputs ...................... 20
3.4 Output in contiguous layout ............................................. 20

4.1 Layers of Crackle .......................................................... 24
4.2 Example of cache implementation ...................................... 25
4.3 Extension of cache to include edge weight information .......... 26

5.1 Topology of Random, Small world and Scale free graphs (left to right order) .......................................................... 28
5.2 Degree distribution .......................................................... 29

6.1 Relative distribution of H-block and irregular graph queries based on their duration ..................................................... 43
6.2 H-block $H^n$ consisting of three smaller H-blocks corresponding to $n - 1$ memory level ..................................................... 47
6.3 Flow path of BFS query on an H-blocked graph that involves switching between the red, green and cyan blocks. Also depicted is the corresponding memory layout ........................................ 49
6.4 Flow path of DFS query on a graph and its corresponding memory layout ................................................................. 50
6.5 H-blocking on large nodes and its corresponding “Distance Effect” in the memory layout .............................................. 51

7.1 Sharding of H-blocks ....................................................... 53
7.2 The unsymmetrical and symmetrical blocks are depicted by red and blue boundaries respectively ............................ 54

8.1 Performance of H-blocking based on graph .......................... 56
8.2 Performance of H-blocking based on query . . . . . . . . . . . 57
List of Tables

3.1 Hot cache BFS performance with different layouts 22
6.1 Machine configuration 35
6.2 BFS test query results 35
6.3 DFS test query results 37
6.4 SSSP test query results 39
6.5 Fetches per query for BFS 42
6.6 Fetches per query for DFS 44
6.7 Fetches per query for SSSP 44
6.8 Comparison between Irregular, H-block and Neo4j 45
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>breadth first search.</td>
</tr>
<tr>
<td>DFS</td>
<td>depth first search.</td>
</tr>
<tr>
<td>SSSP</td>
<td>single source shortest path.</td>
</tr>
<tr>
<td>HBA</td>
<td>hierarchical blocking algorithm.</td>
</tr>
<tr>
<td>RDBMS</td>
<td>relational database management system.</td>
</tr>
<tr>
<td>WS</td>
<td>small world graph.</td>
</tr>
<tr>
<td>SF</td>
<td>scale free graph.</td>
</tr>
<tr>
<td>ER</td>
<td>random graph.</td>
</tr>
<tr>
<td>LRU</td>
<td>least recently used.</td>
</tr>
<tr>
<td>FIFO</td>
<td>first in first out.</td>
</tr>
<tr>
<td>LIFO</td>
<td>last in first out.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In recent years, graph structured data are increasingly used in applications ranging from social networks, bioinformatics to the semantic web. In a social network, every person is represented as a node and there are many different types of relationships that can exist which are represented as labelled edges between the nodes. In bioinformatics, graphs are used to understand metabolic chain reactions at cellular level and also interactions between different proteins. Graphs are gaining wider usage as they provide a universal and natural scheme to represent real world knowledge. In many domains of real world applications, modelled information contains various interconnections (represented as edges) which grow rapidly. For example, the number of active users of Facebook networking site has grown to 900 million users and over 125 billion friendships [1], since starting in 2004. The BIND database containing protein interaction networks has grown almost 10 times between 2002 and 2004 and has doubled since then. Also, the size of ASTRAL database containing protein structures has grown 3 times since 2002 [4].

The increased use of graph structured data has led to several new database systems based on graph models which are generally categorized under the NoSQL database paradigm. Some of the prominent graph databases are Neo4j [2], Microsoft Trinity [3] and AllegroGraph [5]. On the other hand, the relational database are still widely used especially in large enterprise
applications where the data has reached its full growth and is not expected to change significantly.

Compared to graph databases, the relational database is slow in executing graph queries as the data is stored differently. Each entity is stored in a table format and in order to add any new relation type, the structure has to be modified. For example, if there is a table of a contact list and another table of restaurant list, a new relation connecting a contact to his/her favorite restaurant can be added in following ways:

1. Altering either of the two table so that there is a new column which points towards the other as foreign key; or

2. Avoid altering either table and instead create a new table containing two columns linking row from contact table to that of restaurant table.

These modifications makes altering a graph difficult in relational database. However, this approach is beneficial as it handles data more efficiently by storing different objects as separate relations. This in turn reduces redundant information which is more likely to occur in large unnormalized relations. Also, relational database do not support recursive relationships unlike graph databases. This is due to the fact that they require complex JOINs which are expensive in terms of performance. However, relational database management systems have several desirable properties that have been refined over decades of research. These include reliable high volume batch data processing and repeated transactions capability. Moreover, several organizations have invested considerable time and money into these traditional database systems. This motivates a research question of how graph data structures can be efficiently stored in a relational database and how can graph queries be optimized by analysing the causes and reducing the latencies that occur at different stages of an execution path of query.
1.1 Aims

The primary objective of this project is to optimize query execution through techniques such as prefetching and caching. Prefetching refers to loading of graph nodes before they are required by a subsequent query. Since data-fetch pattern for graph queries depend on the traversal path, it is more likely that at the subsequent stage of a query, there will be need to load neighbouring nodes of the current node. In such cases, it is beneficial to preload the neighbourhood in order to reduce the number of calls made to database instance. This project will largely focus on one such technique called H-blocking which refers to a Hierarchical blocking algorithm [26] on graph data. This aims to reduce data block faults between any two consecutive levels of a computer’s memory hierarchy.

Whenever an application requests for data from a database instance, it introduces network delay in the case of distributed systems and inter-process delay within a system. In order to reduce this lag, a caching layer was also built which first attends to any node requests before forwarding the request to database.

In order to evaluate the performance of different graph queries using these optimization techniques, a graph-processing layer called Crackle was designed and built on the JVM.

The aim of the evaluation is to assess the impact of prefetching and caching techniques on the performance of some basic graph queries such as Breadth First Search (BFS), Depth First Search (DFS) and Single Source Shortest Path (SSSP). In addition, different types of graphs are used for evaluation, in order to understand how their underlying structure affect the query cost. This in turn determines its performance.

Furthermore, the performance of this system using a relational database is compared with a graph-specific database, with the overall aim to reduce performance gap between the two.
Chapter 2

Background and Related Works

First the differences that exist between relational and specialized graph databases will be discussed in order to understand the key advantages that relational database offer. This will be followed by brief discussion on some of the major schemes to represent graph data on relational model. Following that, some of the main sources of latency in a typical SQL query will be explored along with techniques to reduce them. The final section will briefly discuss related prefetching approaches and distributed storage.

2.1 Relational Versus Graph Database Model

The relational database is designed for mission-critical applications where data reliability is very important. The main features can be expressed through the ACID acronym which stands for Atomicity, Consistency, Isolation and Durability. The data is atomic which means during a transaction (single logical operation), if any part of the process fails, then the whole transaction fails. This ensures that at the data state changes only when a transaction completes successfully. The data is also consistent, which implies that it follows certain predefined rules based on table structure, constraints and triggers. It is also important to ensure that separate transactions do not
interfere and ensure data reliability. Such transactional isolation is achieved through implementation of various levels of concurrency and lock control. Finally, durability ensures that once a transaction has been committed, the changes are secured into non-volatile memory even in case of system crash or power failure. The normalisation scheme provided by the relational database provides reduced data redundancy as it maintains only one version of data at any given time. This in turn reduces the cost of storage as well as the programming workload of maintaining consistency across multiple versions. Also unlike the graph database, it provides a neat separation of concerns by separating the programming aspects of handling data from the optimization aspects of the database. It is based on a declarative SQL language that eliminates side effects during query execution. Due to these desirable properties, the relational database are still widely used.

A range of competing vendors provide off-the-shelf products based on this model, which can be deployed very quickly. Additionally, through parallel database approach, it is possible to utilize multiprocessor architecture to achieve parallel execution of operations such as merge, sort and join. This in turn can ensure high-availability. The relational model also offer the advantage allowing easy implementation of sharding. Sharding is a scheme used to distribute the content of larger databases, whereby sets of data rows are split across different servers ensuring that related rows from multiple tables are co-located for faster graph traversal. Data distribution is more complicated for graph databases. For example, Neo4j uses a central controller to link multiple sub graphs whereas Trinity uses distributed controller for each server.

Graph databases are generally based on the BASE acronym that stands for “Basically Available, Soft state and Eventually consistent”. It is useful in cases where there is no need for high availability and real time transactions (consistency). This comes as a trade-off to its highly scalable characteristic that is useful for start-ups and rapidly growing industries. This contrasts with the relational databases, which tends to be used in data critical applications such as banking, payroll and inventory control systems [10].
makes it important to research on how graph queries can be made more efficient using relational databases.

### 2.2 Graph representation

Before considering various methods to optimize graph queries, it would be useful to understand different ways to store graph data structures using the relational model. Some of the standard ways of representing a graph are through adjacency list, adjacency matrices and edge-lists.

In the adjacency list scheme, a graph is represented by a linked list of nodes where each node has its ID and pointer to next node. In addition there is a list that contains pointers to successors of each node.

```c
typedef struct NODE *LIST;
struct NODE {
    int nodeID;
    LIST next;
};
LIST successor[MaxNumOfNodes];
```

The adjacency matrix represents a graph using a two dimensional matrix where each element indicates an edge between any given row and column which correspond to two nodes.

```c
Boolean edge[MaxNumOfNodes][MaxNumOfNodes];
```

While the former is useful to represent sparse graphs, it is more efficient to use an adjacency matrix for dense graph such that 

$$e > \frac{n^2}{64}$$

where $e$ is the number of edges present, $n$ is the maximum number of nodes and so its square is the total number of binary relations possible in the given graph [11].

The edge-list scheme allows simple representation of graph where binary relations can be added or removed easily. It consists of a two columned matrix indicating an origin and destination node in case of directed graphs. For undi-
rected graph, every edge is treated as bidirectional and one way to represent this is by including reciprocal of every edge in the matrix. Due to the fixed tabular design, it can be easily used in a relational database. Given these benefits along with the fact that the origin and destination nodes of an edge can be indexed, this scheme was chosen for the purpose of experimentation.

2.3 Sources of latency

It is important to understand what constitutes the majority of the processing time of a typical SQL query. The major sources of latency introduced during the execution path of a query can be classified as follows:

2.3.1 Between database and application

When any node information is needed, a query is made to database. Relational databases such as MySQL [6] and PostgreSQL [7] are queried over database connectivity driver such as JDBC in the Java environment. These drivers convert incoming request calls into DBMS’s native procedure calls using socket interface. As such, every call made to a database instance introduces network delay depending on whether the server is deployed locally or is remotely accessed.

Currently, the cost of disk storage and DRAM is getting cheaper per byte of data storage. However, access times and bandwidths are not improving at the same pace [8]. This uneven ratio requires a better approach to managing data storage and cache-aware DBMS (Database Management System) algorithms.

In order to minimize the effect of this delay, a local caching layer can be implemented at the application level. This technique will be discussed in the next section.
2.3.2  Between disk and database

When any information is requested for a node, it is first scanned in the PostgreSQL cache. In Linux, PostgreSQL utilizes the operating system's file caching system based on the Least Recently Used (LRU) scheme. As such, the cache used by PostgreSQL consists of blocks of 4KB, each equivalent to a single page size.

When information for any particular node is not present in cache, there is a cache-miss, which results in data being fetched from disk. This introduces disk latency as follows [13]:

\[
\text{Latency} = \text{Seek time} + \text{Rotational delay} + \text{Transfer time} + \text{Controller delay}
\]

Where, seek time is the amount of time it takes for disk arm to physically move from its current track to the track where requested data is stored. The rotational delay is time taken for the disk to rotate the sector containing requested data to the position of disk arm. Transfer time refers to the I/O path delay of transferring data from the disk to the main memory. The disk controller device which controls disk internally also introduces delay. The total delay would be in the range of about 5-15 milliseconds.

As we will see later, the delays introduced from seek and rotation times can be mitigated by ensuring more efficient data layout through Hierarchical blocking. We will also see how the overall delay can be reduced by ensuring better prefetching mechanisms. The main idea is to reduce the number of calls to the disk, and since the transfer rate is usually much faster than disk read, it is more efficient to read more data at each disk access. This would allow maximum utilization of disk during each read. However, it is important that only relevant data is prefected, otherwise there is a risk of introducing wasteful latency without the data being used at the application level.
2.4 Reducing latency

From the discussion in previous section, two main techniques were identified to optimize graph queries:

2.4.1 Caching at Crackle level

The word Crash will be used as short term to refer to Crackle cache. The purpose of Crash is to store node information in low latency access memory. Since a separate cache is maintained by the database, it is important to ensure that the application cache or Crash maintains only relevant node information. The size of Crash is set so that it can contain information of atleast the number of nodes that are covered in domain of any given graph query. The Crash consists of two components, namely the Node List and the Edge List.

The Node List has a length equal to the maximum number of nodes in the graph, whereas the length of the edge list is restricted to an upper limit. The idea behind this scheme is that, given a large enough size of main memory (above 2GB), it is always possible to maintain a pointer list of the size of graph where each element corresponds to a node. The Node List consists of a list of pointers which point to where the information related to node is stored. This information may or may not be loaded into the Crash. If information related to a node is not loaded then it has a null value in the Node List.

The basic implementation consists of a Node List pointing to location in the Edge List where neighbours for any given node are stored. Due to the fixed length of the Edge List, it can maintain the edge information of only a certain number of nodes. When there is an overflow, the nodes are ejected from the list based on the FIFO scheme. The list is designed to be tightly packed by storing node information back to back using an array structure. This reduces latency arising out of random access patterns in main memory.
2.4.2 Efficient prefetching

By preloading relevant node information into the Crash layer, the results can improve on two levels. Firstly, it will reduce the network latency with lower calls to database instance. Additionally as a trickle down effect, this will reduce the number of disk access.

The prefetching technique can be beneficially exploited based on the fact that most of the graph query traversals depend on either on semantic or topological locality. As such, it is beneficial to preload the neighbourhood of a visited node in order to improve traversal speed.

Let us consider an example of a road network as shown in Figure 2.1. Typical query on such graph could be “Find local post office” or “Find local restaurants that my friends like”. To process such queries, a set of neighbouring nodes of a certain start node can be preloaded. The neighbourhood can be confined within a certain block whose size can be determined based on heuristics such as geographical bounds based on latitude and longitude, or the physical distance from the central node. Such prefetching techniques are domain dependent as they depend on the semantic knowledge of the underlying graph. However, in this project a different type of mechanism is explored namely Hierarchical Blocking which is domain independent.
2.5 Prefetching and semantic locality

Broadly speaking there are two ways to achieve data prefetching. The first is software controlled where data is prefetched through an instruction set. It is crucial to ensure that the additional overhead introduced by prefetch instruction should not outweigh its gain. This can be achieved thorough automated insertion of prefetching instructions into code to load relevant data based on the program’s access pattern [28, 29]. It is worthwhile to note that prefetching does not reduce the access latency, rather it overlaps it with the computation thereby reducing the effective duration of a program.

The second way to achieve prefetching is through hardware-based schemes. Whenever there is a cache miss for read instruction, its execution is stalled till the required data is fetched from lower cache level. This problem can be addressed by non-blocking cache [30, 31]. In this case, the processor proceeds with subsequent instructions till it reaches the point where the previously requested data is needed [32]. This gives extra head-room for cache to fetch data concurrently. Another hardware technique is by implementing stream buffer that buffers prefetched data seperately from cache [33] and helps in reducing miss rate at the top leve of cache hierarchy.

Similar works to hierarchical blocking have been researched. For example, the FAST algorithm [34] proposes an architecture sensitive layout scheme for B-trees on disk-based database system.

This project will be use edge between nodes to derive the notion of semantic locality which is independent of its domain. This semantic locality can be further exploited for partitioning data into separate components as proposed in SPAR [15] where direct neighbours are co-located in same server. Another example of data partitioning is the Cassandra which distributes data storage across a cluster using multi-dimensional map [14]. Storage systems such as Ficus [16], Farasite [17] and Coda [18] distribute multiple replicas of file across the clients.

However, these systems can be leveraged by using semantic knowledge to group
related data locally which would in turn reduce the processing overhead of accessing multiple servers. It is crucial to understand the topology of graph and how it affects different graph queries. The semantic knowledge can be used in online social networks such as Twitter, Facebook and Google+ to improve the system efficiency [20]. The locality principle can be exploited to improve storage efficiency and processing speed in distributed storage systems.
Chapter 3

Hierarchical Blocking

A typical modern computer system contains various levels of cache to store data to form a memory hierarchy as shown in Figure 3.1. At top region of the hierarchy are CPU registers and caches such as L1 & L2 whereas at the bottom of the list are non-volatile memory storages such as hard disks. As we move up the hierarchy, the memory gets closer to processor and hence becomes faster. However, the size of cache decreases with increasing level. This results in a trade-off between miss-rate and latency. The high level caches have lower latency compared to those at lower level but due to size limitation, there is a higher miss-rate. In case of a miss, the data has to be fetched from subsequent cache below the current level.

Data at each level is arranged in form of blocks of fixed length. When any specific data is fetched by a higher cache, the whole block containing the requested data is loaded. If there is no space left in the higher cache then one of its existing block is evicted.

Generally, a CPU cache is made up of blocks of length 64 bytes known as cache lines. If the processor requests for certain data and there is a miss from the CPU cache, the data is then fetched from the main memory. The Translation Lookaside Buffer (TLB) is then accessed to load data from the main memory. The TLBs have lower latency compared to main memory and
contain mapping of logical to physical address for 4KB blocks in the main memory.

Figure 3.1: Memory hierarchy of a typical computer system with associated access cycles for each level

The number of cache misses can be reduced by ensuring efficient spatial locality within each data block such that neighbouring cells in a block contain data of higher relevance.

The van Emde Boas (VEB) tree [27] was introduced as a data layout structure for trees. It arranges the nodes of a tree to increase spatial locality independent of how the cache hierarchy is organized in a machine. The algorithm for VEB takes a tree as input and outputs contiguous blocks data where each block consists of a subtree nodes arranged in Breadth First Search (BFS) order.

For illustration, consider a binary tree is used for VEB layout as shown in Figure 3.2. The tree is then split at a depth of D/2 where D is maximum depth of the tree from its original root. The subtree obtained during the split is traversed in BFS order and forms the first block. The remaining part of tree now consists of new roots \( O(2^{D/2}) \), each of which are then
recursively traversed as separate subtree and then output as separate blocks of contiguous memory layout.

Figure 3.2: van Emde Boas layout scheme

Inspired from this layout scheme, the Hierarchical Blocking Algorithm (HBA) was introduced. Contrary to the VEB algorithm where a splitting point is determined during its execution based on the diameter of a given graph, in case of HBA, we make assumption that information regarding memory hierarchy of a given machine is supplied to the algorithm as input. With the help of this information, the splitting point is determined by comparing the space taken by in-memory representation of vertices traversed with the required size to fit into block size of cache. This split off can be implemented for multiple levels of cache as follows:

1. CPU caches including L1, L2 and L3 which have block size of 64 bytes
2. Main memory (DRAM) with block size of 1024 bytes
3. The TLB caching for address translation of pages of size 4096 bytes
4. The TLB caching for superpage translations where pages are clustered into a superpage of size 2097152 bytes

The HBA blocking works with a slight variation to the VEB algorithm. It takes an arbitrary directed or undirected graph and generates a data layout
in form of contiguous blocks each of which represent a separate sub-graph. It consists of series of repeated breadth first searches at the end of which the leaves are used as roots for new searches.

Let $A(x)$ denote application of an algorithm $A$ on one or more vertex of a graph denoted by $x$. If the algorithm is applied on a list of vertices then the output of every individual vertex is concatenated in the order in which the vertices are passed as inputs. Let $A^k$ denote $k$ recursive application of an algorithm on input such that the output of one recursion is used as input of the next. Let $|v|$ represent the amount of space taken by a vertex $v$ and similarly $|A(x)|$ represent the size taken by the set of vertex generated as output by algorithm $A$ with $x$ as input.

Let $BFS_d(x)$ denote application of breadth first search with vertex $x$ as root and $d$ as the maximum depth to be traversed. The HBA can be denoted as $BFS_d$ as it is essentially repeated BFS with fixed upper bound based on memory hierarchy. The HBA loads all the nodes that it traverses into a spatially contiguous layout. This layout is divided into blocks of size depending on units of each memory hierarchy levels passed as input to algorithm.

Consider a memory hierarchy of $n$ levels with monotonically increasing units of spatial locality denoted as $s_i$ for level $i$, such that $s_i < s_{i+1}$.

Now we can define the HBA as follows:

- $P_1(0) = BFS_d(0)$ where the depth is derived such that:
  - If $|BFS_d(0)| > s_1$, then $d = 0$
  - Else $d$ is derived such that $|BFS_{d-1}(0)| < s_1$ and $|BFS_d(0)| \geq s_1$
- $P_1(0) = P^{k}_{i-1}(0)$ where $k$ is derived such that:
  - If $|P_{i-1}(0)| > s_i$, then $k = 1$
  - Else $k$ is derived such that $|P^{k-1}_{i-1}(0)| < s_i$ and $|P^k_{i-1}(0)| \geq s_i$

The HBA starts from a certain root node $r$ and recursively applies $P_n$ on $r$ where $n$ is number of levels in memory hierarchy. As the algorithm traverses
the tree, it generates memory sensitive layout of the nodes it traverses. It is important to note that the algorithm allows the size of output block to exceed its upper limit of memory unit. This is due to the fact that evaluation of block size takes place only at the beginning of new level in tree, as such the output block size could possibly exceed the memory unit limit (passed as input to algorithm) while at the middle of traversing a tree level.

The algorithm was originally inspired by working with tree data structures as input. But it can be slightly modified in order to work with graph data as well. It is important to keep into consideration that in graphs a child node can have incoming edges from more than one parent nodes. As such, the algorithm needs to ensure that a child node is visited exactly once in order to avoid recursive loop and redundancy. The initial values passed before blocking are shown under Algorithm 1 and the modified Hierarchical blocking is shown under Algorithm 2.

Algorithm 1 Initial variables

**Input:** root as start node

**Input:** $N$ as total number of memory levels

**Input:** $S[1..N]$ containing block size of each memory level

**Initialize:** $S[N+1] \leftarrow \infty$

**Initialize:** roots $\leftarrow$ new array of $N+1$ empty queues

**Initialize:** leaves $\leftarrow$ new array of $N+1$ empty queues

**Initialize:** space $\leftarrow$ new array of $N+1$ elements initialized to zero

**Initialize:** level $\leftarrow N + 1$

The algorithm uses queue structure to store roots and leaves at each levels of the hierarchy. The array space contains the amount of space that the nodes at any given level are taking. As such, the space taken by a particular level $x$ at any given point of time is given by $space[x]$. This value is then compared with the limit set by the array $S$ passed as input containing upper bound for all levels. When the upper bound is reached, the algorithm starts a fresh block by taking the leaves of previous block as roots for new blocks.

The steps 31-33 of Algorithm 2 implement $BFS_d$. First the node is checked for presence in the layout in order to prevent duplication. Note that the
Algorithm 2 Hierarchical Blocking Algorithm

1: roots[N+1].push(root)
2: loop
3: if roots[level] = empty then
4: roots[level] ← leaves[level]
5: leaves[level] ← empty
6: end if
7: if space[level] ≥ S[level] then
8: leaves[level + 1].push(roots[level]) ▶ Promoted to higher level
9: roots[level] ← ∅
10: space[level + 1] ← space[level + 1] + space[level]
11: level ← level + 1
12: CONTINUE loop
13: end if
14: if roots[level] = ∅ then
15: if level = N + 1 then
16: EXIT loop
17: else
18: space[level + 1] ← space[level + 1] + space[level]
19: level ← level + 1
20: end if
21: else
22: node ← roots[level].pop()
23: if level > 1 then
24: roots[level − 1].push(node)
25: space[level − 1] ← 0
26: level ← level − 1
27: else
28: if node ∉ output then
29: output.append(node) ▶ Node copied to H-blocked layout
30: space[1] ← space[1] + sizeof(node)
31: leaves[1].append(node.childNodes())
32: end if
33: end if
34: end if
35: end if
36: end if
37: end loop
step 30 ensures that a node is checked before being copied to output layout. This can be achieved in a more efficient manner such as by implementing a set containing the node number for all loaded nodes. The layout output needs to be stored in contiguous data structure that provide physical locality such as an array or buffer. The child nodes at the end of BFS are added to $leaves[1]$. If the amount of space used by nodes at level 1 is less than the unit of spatial locality for hierarchy level 1, then all the leaves of the BFS are moved to $roots[1]$. The algorithm then performs new BFS on each of these separately as new roots to uncover their child nodes. As such, only when the total space occupied by nodes at the lowest level is equal to or greater than the unit of spatial locality at the lowest level, the level variable is bumped and the leaves it generated are moved to the next higher level of 2.

As such, the splitting point is determined dynamically during the algorithm execution and so is the depth $d$ for operation $BFS_d$ fixed based on same principle.

For any level greater than one, the remaining steps apply which implement $P_i$, where $P_i$ is input of list of nodes stored in root[$i$]. Steps 7-12 of Algorithm 2 checks whether or not the space taken by block for level $i$ due to repeated application of $P_{i-1}$ has exceeded the unit of spatial locality based on $s[i]$. If it has exceeded the limit, then the output leaves are passed to the next higher level $i + 1$. Otherwise, the operation $P_{i-1}$ is repeatedly called on the head nodes of the $roots[i]$ list. The final level of $n + 1$ is designed as simply a placeholder for the final output of $P_n$ and is initialized with infinite size limit.

Let us consider a simple example with an input graph and memory hierarchy as shown in Figure 3.3. The memory hierarchy consists of two levels. The first level corresponds to the Dynamic Random Access Memory with block size of 1 KB. The second level corresponds to the Translation Lookaside Buffer cache with block size of 4 KB.

The HBA would begin from the node $a$ as root which will be added into $roots[n + 1]$. The node passes through numerous iterations and finally gets
passed to roots[1] where it gets copied into output block B1. The child nodes b
and c are then added into leaves[1]. As roots[1] is empty, they are transferred
to this queue. Lets assume the case where S[1] < space[1] which means that
the size of B1 so far is within the size of DRAM block. In such case, the
HBA will continue at same level and add nodes b and c into B1. After
this the nodes d, e and f will be added into roots[1]. Now let's assume the
case where S[1] ≥ space[1]. The nodes d, e and f are then promoted to
level 2 and P2 is applied on them individually to give outputs blocks B2,
B3 and B4 respectively. At the end, the child nodes of nodes g, h, i, j, k,
and l will be added into roots[1]. Assuming that S[1] ≥ space[1] for blocks
B2, B3 and B4, these nodes will be promoted to level 2. However, now the
has surpassed the size of TLB cache block. As such the nodes in roots[1] are
further promoted to level 3 and P3 blocking is applied on them individually.
The layout output is shown in Figure 3.4.

Figure 3.3: Undirected graph and memory hierarchy inputs

Figure 3.4: Output in contiguous layout
3.1 Hot cache analysis of blocking effect

In order to understand how performance is affected by the H-blocking scheme, preliminary tests were conducted. In these tests, an original graph generated from the Stanford Network Analysis Package (SNAP) generator was loaded into memory and stored in a contiguous array using Java’s ByteBuffer construct. The nodes were stored in the increasing order based on their node number. The in-memory representation of the graph was then modified to following three different layouts:

1. Irregular
2. H-Block
3. BFS

In the irregular layout, data was scrambled in order to ensure that there is no inherent locality between parent and child nodes. This mimics real world scenario where graph nodes are dynamically modified thereby lowering the locality of nodes based on their node number. The second layout was generated by applying the H-blocking procedure which is designed to improve spatial locality of parent and child nodes in array by improving their locality based on node number. The last layout used for comparison was from BFS scheme. Here the graph was traversed in Breadth First Search (BFS) manner and the node numbers were rearranged based on the traversal order.

Although H-blocking is essentially repeated application of BFS, there is a key difference from regular BFS. The H-blocking procedure applies BFS only to a certain depth until a splitting point is reached. After that it continues series of fresh BFS on leave nodes of previous search. As such, it is not fully optimized to BFS query. It has an element of both parent-child as well as inter-sibling locality, with the main objective to create blocks compatible with the memory hierarchy of system. The BFS results on each of the three in-memory layouts are as shown in Table 3.1.
The test was applied on a random graph of size 10 million nodes. The BFS query covered the complete graph. As expected, the BFS layout gave highest performance followed by H-block. It took about 17.57 sec which is almost 2x increase compared to the irregular layout at 34.28 sec. The BFS layout gave slight better performance compared to H-block of about 10% which is due to the fact that BFS layout is ideal for the BFS query. The main cause of disparity between the H-block and BFS layouts can be explained based on the “Distance Effect” which is discussed in detail under section 6.4.3. The idea is that as we move down the graph, the number of nodes at any given BFS level gets larger. This increases the spatial distance between sibling nodes in the H-block layout. In contrast, the BFS layout does not suffer from this effect as it stores the nodes for every level linearly in the order of BFS traversal. Due to this, the number of blocks that need to be fetched from DRAM into CPU cache is higher in case of H-block compared to BFS layout which affects its relative performance. However, it is worthwhile to note that the H-block is expected to give better generalization over range of other queries such as Depth First Search (DFS) and Single Source Shortest Path (SSSP).
Chapter 4

System Overview

The experiments were carried out using Crackle which is built on the JVM environment. Crackle consists of four layers as shown in Figure 4.1. The individual layers are as follows:

1. Query processing: This layer has implementation of graph queries such as BFS, DFS and SSSP. It is closely coupled with the caching layer and calls it whenever any node information is needed.

2. Caching: This layer implements an array that contains node information. Alongside, it maintains a list of all the nodes that have been loaded. When the query processing layer requests for a node, it is first checked within local cache. If there is a miss, then this request is forwarded to the prefetching layer which loads data from the database.

3. Data loading: This layer implements SQL calls to database. The required data may already be present in the database cache or else it needs to be loaded from disk. This also accomplishes prefetching which is done implicitly in case of H-block layout.

4. Database: This refers to data source used to store and query graphs. There were two databases used in my experiments. The first one was PostgreSQL which is based on relational model. The second data source used was Neo4j which is graph database and uses key-value stores.
PostgreSQL was used for evaluation as it is a standard open source database available based on the SQL standard. It also supports wide range of data types and SQL queries. The Neo4j was used as the other database for testing as it is a popular NoSQL graph database and runs on the JVM.

4.1 Caching

We had discussed earlier that every call made to the PostgreSQL introduces latency. As such, it becomes more beneficial to maintain a separate cache at the application level as subset of the PostgreSQL cache. The aim here is to reduce the calls made to database instance by increasing data availability at application level.

The cache is designed based on the basic idea that it is always possible to load the node-list into memory but not to load the complete edge-list. This is due to the fact that the number of nodes is lower compared to the number of edges in a connected graph. The node-list is implemented as an array and each cell is mapped to a node based on its node number also referred as node ID.

The second data structure used is ByteBuffer which stores node information of all the loaded nodes. It has an upper limit as only a fraction of node from graph can be loaded at any given time. It stores information for each node in
form of blocks. Each block begins with node ID to which it belongs followed by ID of its child nodes. The blocks end is marked by a delimiter -1. Each unit within a block has size of 8 bytes and the total block size depends on degree of node.

A write pointer is maintained which keeps track of where the last write was done. Whenever a new node is loaded into cache, the write pointer is incremented to the end of the newly loaded block. When the pointer reaches end of ByteBuffer, it restarts from the top removing the oldest block in FIFO manner. The node list contains pointer to the location where a node is loaded in ByteBuffer. If a node is not loaded then the cell takes the value of -1.

![Figure 4.2: Example of cache implementation](image)

In Figure 4.2, the nodes 1 and 2 are loaded at position 0 and 3 respectively. On the other hand nodes 3 and K are not loaded and have value of -1 in the node list.

It is worth noting, that the ByteBuffer implementation can easily be extended to add any additional information of node. For example, edge weights can be introduced after every child node as shown in Figure 4.3.

As such, the cache has been designed to ensure contiguous allocation of memory such that there is single pointer for each node pointing to its dedicated block. By using primitive data structures, it is possible to precisely predict and minimize the amount of space the cache takes. It is given by following
equation:

\[ \text{Cache size} = (\text{Graph size} \times 8) + \text{Maximum number of nodes to load} \times \text{node size} \times 8 \]

In the above equation the size of node depends on its degree. For a simple implementation that does not take into account the edge weights, it is given by \( \text{node size} = \text{degree} + 2 \). Here, the degree is added with 2 which corresponds to the start and end limiters present for each node. The node size can be adjusted to accommodate greater information for each node.

Different graphs can have different degree distribution. For example, scale free networks follow power law distribution with presence of nodes with very high degree. In such cases the average degree does not give good heuristic of how much space a node could assume. As a good design principle, it is important to ensure that the ByteBuffer implementation can work even in worst case scenario i.e.- can load node with highest degree. Since any given graph can have a node with degree as high as \((N-1)\) where \(N\) is total number of nodes, the ByteBuffer must have size of at least \((N - 1 + \text{limiter}_{\text{START}} + \text{limiter}_{\text{END}}) \times 8\) bytes so that it can load the largest node present in graph at any given time.
Chapter 5

Data Preparation

The graphs used for the experiments were generated using the SNAP. The generated graphs were of three different types as follows:

1. Small world (WS) graph with 10M nodes and 60M directed edges based on the Watts Strogatz model
2. Random (ER) connectivity graph with 10M nodes and 60M directed edges based on the Erdos Rnyi model
3. Scale free (SF) network graph with 10M nodes and 30M directed edges based on the Barabasi Albert model

In the Watts Strogatz model, all the nodes are first arranged in a regular lattice. A fraction of edges are chosen and their endpoints are repositioned based on rewiring probability. Some of the other parameters taken into account are the diameter which is typically fixed to around 6 hops and Poisson distribution of degree with each node having greater than three local neighbours. Typically it consists of a single giant component due to high degree of clustering.

The Erdos Rnyi model has two variants. In the first case $G(n, e)$, a graph is uniformly chosen at random from all the possible graphs with $n$ nodes and $e$ edges. In the second variant $G(n, p)$, two nodes are randomly connected with
probability $p$, which is independent of other edges. In this case the average degree is given by $K \sim n \ast p$.

The Barabasi Albert model uses preferential attachment mechanism to generate a scale free network graph. The algorithm begins with a certain number of nodes, each having at least one neighbour. As new nodes are added, they are connected to the existing ones where the probability of a new edge to a node is directly proportional to its degree. As such, certain nodes end up with very high degree as hubs. Power law gives the distribution of degree as $P(k) \sim c \ast k^{-\gamma}$, where $c$ is a normalization constant and $\gamma$ is power law degree exponent usually in the range of $2 < \gamma < 3$.

The underlying structure of each graph is shown in Figure 5.1 and their corresponding degree distribution is displayed in Figure 5.2.

![Figure 5.1: Topology of Random, Small world and Scale free graphs (left to right order)](image)

### 5.1 Graph pre-processing

Each graph was pre-processed to generate its two versions. The first version was generated by applying H-blocking algorithm which gives a memory efficient H-block layout. The second version was generated by randomizing the graph through shuffling of the node ID as key.
The generated graph consists of an edge list containing the node IDs of source and destination. The graph initially has locality depending on model used by the generator as described in the previous section.

The graph is then passed to an H-blocking application implemented in Java. The application traverses input graph in a series of repeated breadth first search as described in the earlier section on H-blocking. The memory hierarchy information is also passed as input. Maintaining focus on improving the efficiency between the disk and main memory, the blocking was implemented only at one level for VM pages of size 4KB. A low degree node was chosen as start node to minimize offset in layout. This ensures that there is lower overshoot of block sizes. On the other hand, nodes with high degree are more likely to overshoot the upper limit of block size set by spatial units of memory.

During the execution of blocking application, the space taken by a node with $n$ number of child nodes is calculated as $n \times 2 \times 8$. Where 8 bytes is the size taken by a node number (represented as an integer), multiplied by 2 for each source and destination node in an edge list. Each node is visited only once.
by the blocking algorithm.

The blocking application traverses the graph and generates a mapping vector as output that maps original node number to its new blocked number. The whole idea is to layout the edge list contiguously based on the new node key scheme which would give more efficient blocking sensitive to spatial units of memory. The mapping vector is applied to the original graph to generate more efficient H-blocked version of the graph. This was achieved through following steps:

1. Import the mapping vector to database. It would be stored as a table with two columns each corresponding to old and new node ID

2. The new H-blocked graph is then generated through two inner joins as follows:

   (a) Inner join between the mapping vector and original graph so that the ID of source node is mapped to its equivalent new ID:

   CREATE TABLE tmp AS (SELECT m.ID AS n FROM map AS m INNER JOIN graph AS d ON d.n = m.ID);

   Here, here map refers to the H-block layout mapping vector, tmp is intermediate table created, graph refers to original graph table, n refers to source node.

   (b) Similarly, an additional join is performed to get final H-blocked graph:

   CREATE TABLE graphHB AS (SELECT d.n AS n FROM map AS m INNER JOIN tmp AS d ON d.n = m.ID);

   Here graphHB refers to the newly H-blocked graph.
3. When \textit{graphHB} is generated, its rows are not stored in any particular order. In order to organize it in right format, the \textit{nFROM} and \textit{nTO} columns are indexed.

4. The \textit{graphHB} is then reorganized by taking the index on \textit{nFROM} as key. This is achieved by the \textit{CLUSTER} function in PostgreSQL:

\begin{verbatim}
CLUSTER index_{nFROM} ON graphHB;
\end{verbatim}

Here \textit{index_{nFROM}} is index on the \textit{nFROM} column of \textit{graphHB}.

The clustering ensures that the edge list is physically stored based on the order of node ID. As such, nodes with closer ID share greater spatial locality.

5.1.2 Randomization

Earlier we noted that the graphs output by generator are basically derived from a lattice form. As such, the nodes have some inherent locality so that nodes with closer ID are likely to have greater preferential bonding between them. In order to have clear benchmark it is important to make sure that the graph used for comparison does not have any particular inherent locality. In order to achieve this, the graphs need to be modified to generate its new randomized copy. This process is similar to how the H-blocked copy is generated.

1. A scrambling vector is generated that contains mapping of original node id to a new random ID. The vector is checked not to contain any collision.

2. The randomization is achieved in form of two inner joins as follows:

   (a) Inner join between graph and scrambling vector \textit{m}:

   \begin{verbatim}
   CREATE tmp AS (SELECT m.IDOLD AS nFROM, d.nTO AS nTO
   FROM map AS m INNER JOIN graph AS d ON m.IDOLD =
   \end{verbatim}
Here \textit{map} refers to the shuffling map vector.

(b) Inner join between intermediate table (tmp) and scrambling vector:

\begin{verbatim}
CREATE graphR AS (SELECT d.n_{FROM} AS n_{FROM}, m.ID\_NEW AS n_{TO} FROM map AS m INNER JOIN tmp AS d ON m.ID\_OLD = tmp.n_{TO});
\end{verbatim}

Here \textit{graphR} is a newly generated random version of the original graph.

3. The two columns of graphR are indexed and clustered to rearrange the data entries based on increasing order of node ID:

\begin{verbatim}
CLUSTER index.n_{FROM} ON graphR;
\end{verbatim}

Here \textit{index.n_{FROM}} is index on the \textit{n_{FROM}} column of \textit{graphR}.

At the end of pre-processing, each graph generates two versions: H-blocked and irregular (randomized). This step is applied to all the three different types of graph used in experiments. The two versions provide a basis for comparison in the evaluation section.

Since the layout is implemented at the disk level, every time the database needs to load any particular node, the whole block containing the node is loaded into memory. This block contains the requested node’s neighbourhood. In this way H-blocking achieves implicit prefetching. It saves the cost which would otherwise occur in case of prefetching techniques that make use of explicit calls. Other major benefit that the H-blocking provides is that it improves the spatial locality of neighbouring nodes. This compaction takes into consideration all the levels of memory hierarchy leading to most efficient utilization of space at every memory level.
Chapter 6

Evaluation

The experiments were conducted in two batches. The first batch of tests was carried out in order to analyse the performance of every graph with test queries using Crackle. The second batch of tests were conducted on the best performing graph from the previous batch in order to gain an in-depth analysis of individual test queries by tracking the block faults at different cache levels of the PostgreSQL Buffer and Kernel file system cache. The aim of the first batch is to build an overall understanding of how H-blocking influences test query performance on different graphs. The second batch tests were conducted to build from the first batch and narrow down the analysis by observing the influence of H-blocking by tracking cache utilization. The final part of the second batch includes comparison tests between H-blocked relational database and Neo4j. The configurations of machine used for evaluation are as shown in Table 6.1.

6.1 Batch 1

The runs for each query were conducted with ten iterations. The idea was to have each test covers at least 10k nodes with varying centrality and to run them with different starting points. Only exception is WS with BFS where
the number of nodes covered is small, however, further tests were carried out with much larger coverage of about 10M nodes in the second batch tests. The performance of every individual run was averaged to get the overall performance for each test query. The aim of this batch is to evaluate the output results for individual graphs independent of each other. As such, the search space for each graph need not be equal.

### 6.1.1 Breadth First Search

In the first set of experiments, the BFS query was tested on different graphs. For each graph, the performance was compared between its H-blocked and irregular layouts. The results of cold-cache BFS query for different layouts are shown in Table 6.2.

<table>
<thead>
<tr>
<th></th>
<th>WS</th>
<th>SF</th>
<th>ER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Central</td>
<td>Edge</td>
<td>Low</td>
</tr>
<tr>
<td>Irregular (sec)</td>
<td>4.12</td>
<td>2.11</td>
<td>73.37</td>
</tr>
<tr>
<td>H-Block (sec)</td>
<td>3.51</td>
<td>1.08</td>
<td>60.42</td>
</tr>
<tr>
<td>Query size (nodes)</td>
<td>4K</td>
<td>0.9K</td>
<td>76K</td>
</tr>
<tr>
<td>Gain (%)</td>
<td>14.65</td>
<td>48.82</td>
<td>17.66</td>
</tr>
</tbody>
</table>

Table 6.2: BFS test query results
Small World

The BFS was tested on different types of start node. In case of the small world graph, tests were run separately on nodes with high and low centrality. This enables us to understand the impact of performance based on centrality of start node of traversal. As can be seen in the results, the performance improved by about 14.65% with HBA in case of more central start nodes. On the other hand, the start nodes that have lower connectivity (at outer bounds of clique-like cluster) showed improvement by about 48.82% which is above 2x gain in some cases. The difference between the central and edge node performance can be explained based on the total number of nodes visited. In case of edge nodes, the BFS tree level grows by rate of around 3 degree whereas in case of central nodes it grows with around 5 degree. This fact is also evident by the total number of nodes visited in the former case as high as 9k compared to latter at 40k. As such, the performance between them would converge with larger query to baseline improvement of about 14.65% as we will see in the second batch tests.

Scale Free

In the case of graph with scale free characteristic, the start nodes were divided based on their overall degree: Low (< 6), Medium (> 1000 and < 10K) and High (>10K). The nodes were chosen with random distribution in order to maximize separation between the nodes. The performance improvement of medium and low start nodes were 21.13% and 17.66% respectively. The gain of high degree node was 13.57%. While the tests on low degree covered about 7.6k nodes per query on average, the other start nodes covered well above the half of all nodes in graph. As such, while the low node results can slightly vary, the result of other two are fairly stable given the large fraction of graph it traverses. It is worthwhile to note that the percentage wise performance gap between high and medium node of approximately 7.5% is due to the fact that unlike medium node, in case of high degree node, the BFS was restricted to maximum of two hops. This is a reasonable assumption since
the high degree node had high centrality and covered almost 64% of the graph.

**Random**

The final tests were conducted on random graph which have relatively low degree distribution compared to other two graphs. As there is no particular underlying structure and each nodes connectivity is derived randomly based on probability, the start nodes were also chosen randomly. The tests covered a total of 10K nodes with HB layout showing speedup by around 25.9%. It is interesting to note that the performance gain is high in this case and comparable to that of WS. This can largely be explained due to the fact that the graph inherently lacks any form of spatial locality based of nodes and as such HB layout is expected to improve the performance.

### 6.1.2 Depth First Search

The second set of experiments was carried with DFS query. Similar to BFS experiments, the performance was benchmarked against irregular version of graphs as shown in Table 6.3.

<table>
<thead>
<tr>
<th></th>
<th>WS</th>
<th>SF</th>
<th>ER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Central</td>
<td>Edge</td>
<td>Low</td>
</tr>
<tr>
<td>Irregular (sec)</td>
<td>40.6</td>
<td>42.43</td>
<td>35.37</td>
</tr>
<tr>
<td>H-Block (sec)</td>
<td>5.21</td>
<td>5.06</td>
<td>27.79</td>
</tr>
<tr>
<td>Query size (nodes)</td>
<td>10K</td>
<td>10K</td>
<td>10K</td>
</tr>
<tr>
<td>Gain (%)</td>
<td>87.1 (≈8x)</td>
<td>88 (≈8x)</td>
<td>21.4</td>
</tr>
</tbody>
</table>

Table 6.3: DFS test query results
Small World

The tests that were carried out on small world graph covered about 10k nodes. Keeping in line with BFS benchmark, the start nodes were classified as central and edge nodes. The tests were then carried out separately for each case. The results in both cases showed performance gain in the region of 87-88%. This almost 8x improvement over the standard case, shows the great efficiency HBA can introduce in queries that have vertically oriented traversal such as DFS. Another example is of A* where the query is designed to traverse as deep as possible to a sufficient level before it branches out for more horizontal search. It is also interesting to note that there is little influence of how degree affects the final performance. Again, this is due to vertical traversal of DFS where the number of child nodes at any particular level has little effect compared to number of maximum hops for any given start node. This was restricted to 1K in our experiments.

Scale Free

The second tests were carried with scale free graph with start node classifications: low, medium and high. The performance gain is consistent for low and high nodes in the range of approximately 20-21%. The observation for medium node showed relatively lower speedup by approximately 5%. This can partially be explained due to the fact each query visits about 1k unique nodes with no predefined path. A path that traverses over a large node means that larger blocks need to be loaded to memory. This lag is sufficient to slightly vary performance that in this case came to be around 2 seconds between medium and high nodes. This lag becomes noticeable when a large node appears somewhere at the tail of a DFS search as they are first to be revisited compared to nodes visited at beginning of path. This is unlikely to occur when the start node itself is of high degree. Clearly, the difference based on start node would converge with increase in the coverage of query. This is due to the fact that large queries would have fair share in visiting large nodes.
In order to take a deeper look, a new set of tests were implemented which covered large fraction of nodes which constitutes about 10% of the graph. The nodes were chosen with random distribution of degree. The results showed similar trend with about 20.65% gain. Furthermore, there was a decrease in the number of page miss at PostgreSQL cache level. This was reflected by decrease in number of tuple fetch from disk to about 1.4M for H-block graph compared to around 6.6M fetches in case of irregular graph. This corresponds to 5177 fewer fetched per query on average with H-block graph.

Random

The last tests where carried with Random graph where the start nodes were chosen with random distribution. The DFS search showed an improvement by about 57.96% that is gain of 2x moving from irregular to H-blocked format. As there is no community structure in this graph, the edge connectivity between nodes is scattered which in turn makes a vertical search inefficient with the non-optimized graph layout.

6.1.3 Single Source Shortest Path

The third set of tests was carried out with Dijkstra Algorithm to find shortest path between a pair of nodes. The pairs of connected nodes were randomly selected to run SSSP query and in overall around 100k nodes were covered by the tests. The test results are shown in Table 6.4

<table>
<thead>
<tr>
<th></th>
<th>WS</th>
<th>SF</th>
<th>ER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irregular (sec)</td>
<td>99.37</td>
<td>174.88</td>
<td>175.76</td>
</tr>
<tr>
<td>H-Block (sec)</td>
<td>25.10</td>
<td>174.46</td>
<td>172.34</td>
</tr>
<tr>
<td>Query size (nodes)</td>
<td>100K</td>
<td>100K</td>
<td>100K</td>
</tr>
<tr>
<td>Gain (%)</td>
<td>74.75 (~4x)</td>
<td>0.24</td>
<td>1.95</td>
</tr>
</tbody>
</table>

Table 6.4: SSSP test query results
In small world graph, the HBA showed improvement by 74.75% which is almost 4x speedup case. On the other hand, the performance gain on random graph is 1.95%. The scale free network showed minute gain of 0.24%.

One way to explain the disparity in performance is based on the structural properties of the graphs. The small world consists of small clusters of nodes that have greater internal clustering that give rise to clique-like structures. Such clusters are interconnected through bridges (weak links) between edge nodes while maintaining the law of maximum separation of around 6 hops. In order to execute SSSP between nodes that belong within a same cluster, the query needs to perform Dijkstra algorithm only for member nodes of that cluster assuming non-negative weight. On the other hand, the random graph does not consist of such structure. Due to this, in order to run SSSP between any two random nodes, the shortest path needs to be calculated for a large indefinite number of nodes till the radius of search becomes large enough to reach the destination node. As such, it is relatively slower to run the query on this type of graph. On the other hand, the scale free network follows power distribution of degree. Due to this, running an SSSP query over a very large node results in a greater number of edges being traversed to derive shortest path between any given node pair. This results in relative slowdown in its performance.

Another reason of slow performance with SF and ER can be attributed to the nature of query itself. At each step when a node is visited, the query loads an H-block into memory that consists of the neighbourhood of visited node. However, the next node to be visited may not necessarily be from that block, as the traversal is guided by a priority queue and the algorithm picks a node that has lowest distance (greedy approach). As such, a newly loaded block would be discarded at new step only to be reloaded at later stage when any node from that block is needed. This problem is more acute in case of scale free and random networks where the size of clusters is large. It takes a large number H-blocks to form such large neighbourhood and as only a limited number of H-blocks can be loaded at any given time, only a fraction of neighbourhood is in-memory. This results in greater number of
re-fetching of H-blocks. This renders the purpose of H-block useless, which is to minimize number of fetches.

6.2 Batch 2

In the previous batch of experiments the performance on the small world graph showed consistently good performance compared to other graphs.

Whereas other graphs were traversed to a significantly large number of nodes (above 1M), the tests on small world graph were largely restricted to a maximum of 40k in case of BFS. In order to further examine the performance, a new batch of tests was carried out with small world graph that covered about 7.5% (above 700K nodes).

PostgreSQL buffer and kernel cache

In a regular query, the database first checks the PostgreSQL buffer which is its internal cache. It scans through every row in a relation due to which the cost of a query directly depends on the number of tuples present in a relation[35]. However, the tests were based on indexed source and destination nodes. As such, every query would make an index scan through data pages instead of individual tuples. The optimizer then scan through the large data pages to fetch data tuple. In this batch tests two caching statistics are analysed: PostgresSQL buffer and kernel cache. Whenever there is a miss at PostgreSQL, the database then invokes a call to kernel cache. The reason is because the PostgreSQL buffer is a subset of the kernel cache due to which data absent in the former might still reside in the latter. It is only when a miss occurs at the kernel’s I/O cache, a physical read is required from hard disk. As such, by observing the fetch rates at these two caching layers, it is possible to determine their effectiveness. A lower number of fetch rates would correspond to greater effectiveness as then a cache is able to satisfy large number of requests without redirecting the call to a lower caching layer.
or storage.

6.2.1 Breadth First Search

The first tests were carried out using BFS. The overall performance of H-block was close to 16.11 seconds whereas irregular was at 18.17 seconds corresponding to a performance gain of 11.34.

In the previous batch, it was observed that the edge nodes gave significantly higher gain compared to the central nodes. We had stated that this would converge to its lower range when the search space is increased to cover significant portion of graph and the above results convey that trend.

The performance of the search was evaluated by observing the fetch patterns at different levels such as PostgreSQL buffer cache and kernel cache as show in Table 6.5.

<table>
<thead>
<tr>
<th></th>
<th>PostgreSQL cache (8KB blocks)</th>
<th>Kernel cache (4KB blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H-block</td>
<td>1643</td>
<td>219</td>
</tr>
<tr>
<td>Irregular</td>
<td>2220</td>
<td>798</td>
</tr>
</tbody>
</table>

Table 6.5: Fetches per query for BFS

The number of PostgreSQL fetch is equivalent to misses that occur at this level. The H-blocked showed gain of by about 26%. This translates to decrease in kernel calls and leads to lower latency of inter-process communication, even though it is not significant in comparison to disk access latency.

The kernel cache was calculated by tracking the number of live objects corresponding to relations and index used during scan. Table 6.5 shows the number of 4KB blocks that were read directly from disk to the kernel cache. The overall number of blocks fetched for H-block graph is reduced by about 72.56%. This translates to decrease in the overall number of blocks fetched per query by 579 which reduces latency due to disk access.
6.2.2 Depth First Search

The second tests were carried using DFS. The search was implemented with root nodes with random distribution of degree. The tests covered above 10% of the graph which is slightly above a million nodes.

In the first batch tests, the small world showed marginal improvement by as much as 87.1%. It becomes interesting to check how it performs with more searches that cover nodes of varying centrality. The H-block duration was found to be at 0.95 seconds whereas that of irregular was 1.93 seconds, showing a gain of about 50.78% ($\sim 2x$).

The results show same trend with that of first batch with well above 2x gain. In order to take a close look at the difference between H-block and irregular, Figure 6.1 was plotted which shows the relative distribution of query based on its duration.

![Figure 6.1: Relative distribution of H-block and irregular graph queries based on their duration](image)

One observes that a significant portion (70%-80%) of queries at lower range of 1-3 seconds correspond to H-block with majority within range of 1-6 sec. On the other hand, irregular query appear at long duration range with several of them well above 10 sec.
Further tests were carried out to take a closer look at block fetch patterns at the PostgreSQL and kernel caching layers. The results are shown in Table 6.6.

<table>
<thead>
<tr>
<th></th>
<th>PostgreSQL cache (8KB blocks)</th>
<th>Kernel cache (4KB blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H-block</td>
<td>2942</td>
<td>300</td>
</tr>
<tr>
<td>Irregular</td>
<td>6087</td>
<td>2507</td>
</tr>
</tbody>
</table>

Table 6.6: Fetches per query for DFS

The PostgreSQL cache showed reduction in fetches by about 51.7% (2x). This corresponded to significant reduction in the overall number of blocks fetched from disk to about 88% (8x). As such, the number of fetch calls made to disk is decreased by as much as 2.2K calls. The significant decrease of disk dependency reflects the caching efficiency introduced by H-block layout which in turn accounts for its high performance on DFS query by as much as 8x compared to the irregular layout as observed in the batch 1 tests.

### 6.2.3 Single Source Shortest Path

Final tests were carried out to take a closer look at performance of SSSP query on WS which had shown about 4x gain in previous tests. The tests were carried out with varying path length between a pair of connected nodes. The duration performance showed improvement by 58.72% (2x). The Table 6.7 shows the results where the number of blocks fetched from disk is reduced by 64.28%. On the other hand, the misses at PostgreSQL Buffer cache is reduced by 22 fetches per query. This explains the above 2x performance that has been achieved.

<table>
<thead>
<tr>
<th></th>
<th>PostgreSQL cache (8KB blocks)</th>
<th>Kernel cache (4KB blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H-block</td>
<td>3341</td>
<td>1486</td>
</tr>
<tr>
<td>Irregular</td>
<td>3363</td>
<td>4160</td>
</tr>
</tbody>
</table>

Table 6.7: Fetches per query for SSSP
6.3 Comparison with Neo4j

The final tests were conducted to compare the performance of H-blocked WS graph in relational database with its equivalent irregular version deployed in the Neo4j database. This will give the complete picture of where the optimized RDMS stands in comparison with a specialized graph database. The queries were of equal size in both cases covering about 10% of graph and the start node was selected in a uniform random distribution. The results are summarized in Table 6.8.

<table>
<thead>
<tr>
<th></th>
<th>Irregular</th>
<th>H-block</th>
<th>Neo4j</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS (sec)</td>
<td>18.17</td>
<td>16.11</td>
<td>0.91</td>
</tr>
<tr>
<td>DFS (sec)</td>
<td>1.93</td>
<td>0.95</td>
<td>1.16</td>
</tr>
<tr>
<td>SSSP (sec)</td>
<td>99.37</td>
<td>25.10</td>
<td>11.03</td>
</tr>
</tbody>
</table>

Table 6.8: Comparison between Irregular, H-block and Neo4j

The performance gap between relational and graph database reduced from about 20x to 18x in case of BFS query. On the other hand, the H-blocking surpassed the Neo4j performance by 18% gain for the DFS query. In case of SSSP query, the gap reduced significantly from 9x to about 2x.

6.4 Discussion

6.4.1 Effect of large node on H-block layout

An H-block layout essentially consists of recursive BFS blocks where the number of recursion $N$ is equal to the number of levels in memory hierarchy. As such, a block at any level consists of smaller blocks corresponding its previous level which are arranged in BFS order. For example, the largest block $H^N$ consists of smaller blocks $H^{N-1}$ in BFS format such that the children of leaf nodes of block $H^{N-1}$ form the roots of new $H^{N-1}$ within the $H^N$ block. In this way, the blocks are formed recursively down to the smallest block $H^1$. The size of any block is given by equation 6.1.
\[ |H^n| = |H^{n-1}| \times \frac{S^n}{S_{n-1}} \]  

(6.1)

where \( S^n \) is the size of the \( nth \) level in memory hierarchy. It is important to note that this is only an ideal case relation assuming that \( H^n \) is made up of blocks \( H^{n-1} \) which are all equal in size. In most of the cases, the sizes of constituent blocks vary.

First let us look into a simple case of how \( H^1 \) is formed. During HBA execution, every node is visited in BFS fashion starting from a certain root. The algorithm traverses to a new BFS level only if the size of block traversed is less than the size for first memory level \( S^1 \). As the check occurs only at the end of traversal of any given BFS level, this allows the size of \( H^1 \) formed to exceed \( S^1 \). This introduces offset which increments as we move to higher level.

If the size of first level H-block is \( H^1 \) and size of a single node \(|N|\), then the size of second BFS level in this H-block would be \( H^1 - |N| \). Now in order to consider the worst case scenario, let us consider the case where \( H^1 \) is limiting to \( S^1 \). As such the size of second BFS level can be approximated to \( S^1 - |N| \). Then the number of nodes at this level is given by \( \left( \frac{S^1 - |N|}{|N|} \right) \). There has to be a one node less in order to blocking continue to the third BFS level where only one node is needed to hit the threshold and the rest of nodes in the level introduce offset. This offset is maximized with the maximization of degree at second BFS level. The worst-case scenario is given by equation 6.2.

\[
Offset(H^1) = \left( \frac{S^1 - |N|}{|N|} - 1 \right) \times M_{avg} \times |N| \]  

(6.2)

where \( M_{avg} \) is the average degree of the top \( \left( \frac{S^1 - |N|}{|N|} - 1 \right) \)th nodes with highest degree in a graph.

Let us consider a case a higher level blocking as shown in Figure 6.2. Let there be a block \( H^n \) which comprises of three blocks \( H^{n-1}_a, H^{n-1}_b \) and \( H^{n-1}_c \).
such that the root of $H^{n-1}_a$ is super root (ie- root of $H^n$) and leaves of $H^{n-1}_a$ has two child nodes which act as root of $H^{n-1}_b$ and $H^{n-1}_c$ respectively.

Figure 6.2: H-block $H^n$ consisting of three smaller H-blocks corresponding to $n-1$ memory level

Let's consider a case where the HBA traverse the block $H^{n-1}_a$ and continues to block the children of its leaf nodes separately. In the middle of this, it is possible that the size of $H^n$ can exceed its corresponding threshold $S^n$ as the algorithm checks this only at the end when all child blocks ($H^{n-1}_a$, $H^{n-1}_b$, and $H^{n-1}_c$) have been traversed.

In best case scenario, the size of $H^n$ is exactly equal to $S^n$ and there is no offset. The equation 6.2 for worst case scenario can now be generalized for any H-block level as shown in equation 6.3.

$$\text{Offset}(H^n) \approx \left[ \frac{S^n - |H^{n-1}|_W}{|H^{n-1}|_W} - 1 \right] \times M_{\text{avg}} \times |H^{n-1}|_W \quad (6.3)$$

where $|H^{n-1}|_W$ refers to worst case H-block size for level $n-1$.

The equation shows that the chances of offset depend on memory hierarchy and degree distribution of a graph. The higher the degree distribution and number of memory levels used for blocking, the higher the offset introduced. This is in line with what we would expect.
The offset introduced by a $H^N$ block (the highest level block) propagates to its succeeding block resulting in incremental increase in offset with every block as we move further down the H-block layout. Due to this, the locality of a nodes neighbourhood decrease and it makes H-blocking less efficient whose original purpose is to compact a nodes neighbourhood into same chunk of memory.

This effect is prominent in scale free networks which have nodes with very high degree and to a certain extent on random graphs as well. In order to maintain benchmark, the experiments were carried out using same memory hierarchy across different graphs.

6.4.2 H-block efficiency based on nature of query

Breadth First Search

Let us consider a breadth first search starting from a node A, then the H-block containing this node is loaded to memory. Lets name this as block A. Once the block is loaded, the query traverses breadth wise till end of the leaves at block. Let the leaves have child nodes X, Y and Z. The blocks corresponding to these nodes X, Y and Z are then fetched into memory. The query continues in breadth wise order depicted as black arrows which involves switching from block X (red) to block Y (green) to block Z (cyan) at BFS level as shown in Figure 6.3.

The greater the degree of a node at any given level, the greater the number of switches. Consider a case where a size of an H-block is 4KB (VM page size), then memory utilization to fully load the neighbourhood for a node of degree $n$ is given as $4 \times n$ KB. When the upper bound of allocated memory is reached, some of the existing blocks need to be discarded. Due to this only a fraction of such nodes neighbourhood can be loaded in memory at any given time. This adversely affects a BFS query which involves switching between blocks and since a part of the neighbourhood blocks get discarded for large nodes, these blocks need to be re-fetched when BFS revisits when it moves
Figure 6.3: Flow path of BFS query on an H-blocked graph that involves switching between the red, green and cyan blocks. Also depicted is the corresponding memory layout to its next BFS level.

**Depth First Search**

The H-block format works very well for vertical traversals. Here the same concept of re-fetching applies as discussed in the case of BFS. Re-fetching refers to loading the same block that was previously discarded from memory. Consider a DFS starting from a node that is stored in block B1. The query would load B1 and move down to the furthest node from the start node in block B4 as shown in Figure 6.4. When it completes one complete chain, it revisits the last used block and continues recursively. As such, blocks are revisited in the order in which they were loaded into the memory with the most recently loaded revisited first as shown in black arrow in the figure. Clearly, this is efficient considering a LRU based cache. Due to this, the rate of block-re-fetching is expected to be lower compared to BFS query which
explains significant improvement in performance with the DFS.

Other way to look at this is as depth versus breadth trade-off. The BFS revisits a block in FIFO order whereas DFS does so in LIFO order. Hence with a LRU base cache, DFS has an advantage.

Figure 6.4: Flow path of DFS query on a graph and its corresponding memory layout

**Single Source Shortest Path**

The H-blocking is not very efficient when it comes to traversing graph based on the heuristic of edge distance. But as we will discuss later, the H-block layout can be improved by introducing a better heuristic while blocking.

**6.4.3 Distance Effect**

The HBA showed great improvement as observed in the case with small world graph and to a certain extent on random graphs. However, the scale free graph did not show same change which was due to presence of large degree nodes. One way to tackle this issue is by enforcing a priority queue during
blocking which orders nodes based on their degree. Those nodes that have
degree above a certain threshold which can be deemed as large nodes, can be
pushed into the priority queue. Once the blocking has been completed on the
small nodes, the nodes in priority queue can then be processed in increasing
order of their degree. This ensures that the large nodes are blocked at the end
of layout which in turn minimizes the offset. Large nodes have redundancy
in form of common neighbourhood nodes. If these nodes are processed at
the end, all of its low degree neighbours would have already been blocked
at earlier stage of layout. This is more efficient since then the gap between
parent and child block in layout would be smaller.

\[ \begin{array}{c}
\text{B1} \\
\text{B2} & \text{B3} & \text{B4} \\
\text{B5} & \text{B6} & \text{B7} & \text{B8}
\end{array} \]

Figure 6.5: H-blocking on large nodes and its corresponding “Distance Ef-
fect” in the memory layout

Figure 6.5 shows two pairs of child and parent blocks that are placed distantly
in the layout due to high number of siblings of the child block. The ”Distance
Effect” increases as we move down the BFS levels. As such, the distance
between B4-B8 pair is expected to be greater than the B2-B6 pair. This
effect keeps increasing till the highest-level H-block is formed and blocking
starts anew.
Chapter 7

Future Works

By processing the large nodes at the end, the layout can be improved for queries that traverse in vertical direction such as DFS. It is interesting to note that the BFS is not affected by the Distance Effect and hence the SF in this case showed better performance. As such, there is a trade-off that exists between these two types of query.

The number of memory levels can also lead into increase in the offset. This can be minimized by considering only those levels of memory during block that have significant latency between them for example from disk to DRAM. Some small comparison tests showed that there is isn't any significant difference between HBA performance with 2 and 4 levels. However, more extensive tests can be carried out how the performance is affected with large queries. If the slowdown is significant then it is more sensible to reduce the number of levels. The slowdown can be attributed due to increase in the offset that is introduced in each block whose effect outweighs the benefit of having high memory hierarchy whose original purpose is to reduce the latency between two memory levels.

Following from the distance effect that we discussed, the HBA can be modified so that it is not a pure BFS traversal but also based on other heuristic such as node centrality. This would then allow tuning of the graph between
the opposite extremes of DFS and BFS favouring. There is always a trade-off between the two types of search where tuning a graph for either of them results in less efficient layout for the other search type.

Other possible way to tackle the issue is to instead of processing large nodes at the end, the HBA can be re-implemented so that it checks the size of H-block upon visiting every new node. The HBA used for testing was implemented such that the block size gets checked only at the end of a BFS level which was mainly in order to maintain a symmetric blocking pattern. The negative aspect of this is that it leads to a possibility of introducing offset in each of the blocks. This can be altered by ensuring that a block is allocated whenever its size hits the corresponding memory limit, the offset can be removed.

Let us consider an example as shown in Figure 6.6, where the blue nodes denote a single H-block while the left out nodes D and E are depicted in black which are added as fresh blocks at next iteration of blocking.

![Sharding of H-blocks](image)

Suppose the blocking begins from node A and continues to node C where the block hits the size limit. The traversal should stop there and the remaining nodes D & E should be passed as new roots to next iteration of blocking where they get freshly blocked separately as shown in the above diagram. However, it is important to note that excessive sharing of BFS level can lead to large unsymmetrical blocks as shown in Figure 6.7. As such, this methodology can be applied for only in the case of large nodes.

The sharding can be further modified so that the small degree nodes are
given preference and are visited first. This will ensure that the unvisited nodes on a BFS level beyond the sharding point are of higher degree which are more efficient to be blocked separately.

Lastly, the H-blocking can be modified to implement a different heuristic. The HBA used for evaluation was based on BFS scheme. However, the same HBA can be implemented based on Dijkstra algorithm with heuristic of visiting closest distance node first. Clearly, such a blocking mechanism would perform better for SSSP query in comparison to BFS-based HBA. Similarly, the HBA can also be implemented on DFS query which would give a layout efficient for depth search queries.

What I have demonstrated is that the H-blocking based on BFS can improve the performance of graph queries. This is a baseline model of how blocking can be implemented. If needed, the order of traversal in blocking can itself be modified to tune the final layout it generates for a specific type of query. It would be interesting to compare the performance of different heuristics with the BFS-based HBA as baseline mechanism.
Chapter 8

Summary and Conclusions

The overall performance of individual graphs across different queries is shown in Figure 7.1. The Small World showed highest average gain followed by Random and Scale Free graphs. The Small World showed large deviation due to relatively slow gains observed in BFS compared to other queries. The relative low performance of Scale Free was due to presence of large nodes which introduces offsets in H-blocks and slow down the efficiency of layout. The second contributing factor is that a large neighbourhood increases the number of block fetches which in turn diminishes the gain introduced through better H-block layout. The final factor as discussed in the evaluation section is that due to limited memory size, large neighbourhood lead to greater re-fetching of blocks which adversely affects the performance. In case of Random network, it was observed that the gains were mainly due to increase in spatial locality of neighbouring nodes in an otherwise randomly dispersed nodes topology.

The overall performance of individual queries across different graphs is summarised in Figure 7.2. The DFS showed highest gain followed by SSSP and BFS. The SSSP showed large variation with significantly slow performance in case of the Scale Free and Random graph in comparison to Small World. The disparity was accounted due to difference in graph topology. The other reason for low gain was the nature of SSSP query. The H-blocking is
based on different heuristic compared to SSSP that uses priority queue based on node distance. The DFS and BFS largely gained due to increased parent-child and inter-sibling locality respectively. The gains in DFS was greater in many cases where node degree was small. This allows an H-block to have greater depth thereby favouring vertical traversals.

In conclusion, the H-blocking did show significant improvement in majority of cases and helped close gap with Neo4j for BFS query and surpassed it in case of DFS. The reason for performance gain was further analysed and tracked down due to decrease in miss-rate in database buffer and kernel file system cache which in turn reduced the overall I/O and disk access latencies respectively. In future, there is a scope for further improvement by extending the current H-blocking algorithm using the Crackle interface. This can be achieved by using better heuristics depending on the characterisitics of underlying graph and the nature of graph query against which the blocking needs to be optimized.
Figure 8.2: Performance of H-blocking based on query
Bibliography


