First Tree-like Quantum Data Structure: Quantum B+ Tree

ABSTRACT

Quantum computing is a popular topic in computer science, recently attracting many studies in various areas such as machine learning, network and cryptography. However, the topic of quantum data structures seems long neglected. There is an open problem in the database area: Can we make an improvement on existing data structures by quantum techniques? Consider a dataset of key-record pairs. Given an interval as a query range, a B+ tree can report all the records with keys within this interval, which is called a *range query*. A classical B+ tree answers a range query in $O(\log N + k)$ time, where N is the total number of records and the output size k is the number of records in the interval. It is asymptotically optimal in a classical computer but not efficient enough in a quantum computer, because k (the output size) could be as large as $O(N)$.

In this paper, we propose the *quantum range query* problem. Different from the classical range queries, a *quantum range query* returns the range query results in *quantum bits*, which has broad potential applications due to the foreseeable future advance of quantum computers and quantum algorithms. To the best of our knowledge, we design the first tree-like quantum data structure called the *quantum B+ tree*. Based on this data structure, we propose a hybrid quantum-classical algorithm to do the range search. It answers a static quantum range query in $O(\log N)$ time, which is asymptotically optimal in quantum computers, where, here, "static" means the data is fixed and does not change over time. Since the execution time does not depend on the output size (i.e., k), it is significantly faster than the classical data structure. Moreover, we extend our quantum B+ tree to answer the dynamic quantum range queries efficiently in $O(\log^2 N)$ time. Our experimental results show that our proposed quantum data structures achieve up to 1000x improvement in the number of memory accesses compared to their classical competitors.

1 INTRODUCTION

Consider a dataset where each item in this dataset is in the form of a key-record pair. The range query problem, which is to report all the items with keys within a given query range, has been a longstanding and widely-applied problem. For example, a teacher may need to list all the students who obtained 40-60 marks in an exam, and a smartphone buyer may need to list all the smartphones that sit around $200-$400.

Apart from directly returning all the results to the user, range queries have also been applied as a subroutine of many problems in various areas. One typical example is the recommendation systems [19, 28], where range queries are commonly performed to extract a relatively small set of candidates first from the entire dataset as per user's request and then a recommendation algorithm is executed to find the top recommendations from the candidates.

Consider a movie dataset D containing a list of movies each described as a key-record pair where the key is the release date and the record represents some attributes. Assume that Alice wants to find an interesting movie of 1990s. We first perform a range query of finding a candidate set D' of all the movies with release date from 1990 to 1999 and then give recommendations based on D'.

To answer range queries, many data structures [9, 18, 46] have been proposed in classical computers to store the key-record pairs. One representative and widely-used data structure is the B+ tree [18]. It answers a range query in $O(\log N + k)$ time, where N is the total number of key-record pairs and the output size k is the number of records in the query range. Based on [62], the B+ tree is asymptotically optimal for range queries in classical computers. Note that for all range queries, the linear time of generating all the k results is inevitable, which is not efficient enough when range queries are used as subroutines of algorithms in quantum computers.

Recently, quantum algorithms have attracted a lot of attention. Many quantum algorithms [2, 26, 35, 38, 49, 54] have been proposed and are expected to show quadratic or even exponential speedup compared to classical algorithms, and thus many linear-complexity problems can be solved in sub-linear time in quantum computers.

For instance, a recent quantum recommendation system [35] shows poly-logarithm time complexity to the input size. However, when we consider the scenario where the recommendation is made among the k candidates in a user-specified query range, the potential of quantum algorithms is limited due to the linear time of generating all the k candidates of traditional range queries.

Motivated by the critical limitations of the classical range query in quantum computers, we propose the problem called the *quantum range query*, which returns the answer in *quantum bits*, since we notice that quantum algorithms normally have the input form of quantum bits. Specifically, the quantum range query $QUERY(x, y)$ does not return a list of key-record pairs, but returns the quantum bits in a *superposition* of all the key-record pairs with keys within the query range $[x, y]$, where the superposition is the ability of a quantum system to be in multiple states simultaneously. For example, consider a movie dataset $\{(key_0, rec_0), \cdots, (key_{N-1}, rec_{N-1})\}$, where N is the number of movies. Assume that we can construct a data structure in the quantum computer. If Alice wants to obtain all the movies of the 1990s, a quantum range query $QUERY(1990, 1999)$ will search on the data structure and return the quantum bits in a superposition of all the movies (key_i, rec_i) whose key_i is within the interval $[1990, 1999]$. Motivated by the different scenarios in real-world applications, we study the quantum range query problems in two different cases. The first case is that the dataset is *immutable*, so the problem is called the *static quantum range query*. The second case is that insertions and deletions are supported in the dataset, and the problem is called the *dynamic quantum range query*.

The quantum range queries have many potential applications. (1) In relational database queries, it is common to have a numerical query range as a query condition (e.g., finding the movie with the highest ranking for release date between 1990 and 1999). To execute such a query, a range query is commonly considered for query optimization. Specifically, a range query of retrieving all movies released between 1990 and 1999 is first performed to narrow down the search space to a small subset, and then finding the highest

ranking movie could be executed efficiently on the subset. In the literature, some quantum algorithms [16] have been proposed to support efficient database queries such as finding the database record that has the highest value on a certain field. When enabling query optimization, if we use a quantum range query to obtain the narrowed search space, the result of the quantum range query (in quantum bits) could be applied in those quantum algorithms for fast database queries. (2) In a quantum recommendation system as we mentioned previously, a range query is first used as a filter to find a small set of candidates from the entire dataset, and then a quantum recommendation algorithm [35] is used to make the top recommendation from the candidates. A quantum range query, instead, returns a superposition of the candidates which can be *directly* applied to these *existing* quantum recommendation algorithms. (3) Data binning [21] is a popular approach to improve the accuracy of various machine learning algorithms [12, 60], which applies range queries to group similar items within a query range (i.e., a data bin) together. Applying the quantum range queries, the quantum binning results in superposition can be *readily* used in quantum machine learning algorithms [2].

Besides the above applications, many quantum algorithms are designed to have input and output in the form of superpositions. For example, the HHL algorithm [30] returns the answer to a linear system of equations in a superposition, and it becomes a subroutine of the quantum SVM [48]. Thus, we believe that in the future, more quantum algorithms will benefit from the quantum range queries that return the query results in superposition.

The quantum range query problem has two distinctive characteristics. The first characteristic is that it allows the utilization of data structures for building an index to accelerate database queries, which has been a common approach in the database area. Existing quantum algorithms [14, 22, 26, 27] focus on searching with unstructured data, which can handle small amounts of data but cannot solve the quantum range query problem efficiently in large databases. Among them, the best quantum range query algorithm adapted from [14] returns the result of k items in $O(\sqrt{Nk})$ time, which is inefficient when the dataset size N grows large. Note that although porting the entire database into the quantum computer to build the quantum data structures has an unignorable computational cost, it is considered as a pre-processing step (which corresponds to building a database index) and once this step is done, all the quantum range queries can be answered efficiently.

The second characteristic is the use of quantum computation to handle superpositions efficiently. As mentioned previously, the $O(\log N+k)$ time complexity for a range query in classical computer is already asymptotically optimal, where the $O(k)$ cost of listing out all the k results has no chance to be improved. In the quantum range query problem, however, since we aim to obtain the k results in the form of a superposition, it is possible to eliminate this $O(k)$ cost and thus achieve a better time complexity with the techniques in quantum computation.

Motivated by this, we propose the *quantum B+ tree*, which is the first tree-like quantum data structure to the best of our knowledge. We choose the B+ tree [18] as the first data structure to study in quantum computers. Since the B+ tree is one of the most fundamental and widely-used data structures, it is considered the most suitable one to open a new world of quantum data structures. We design our quantum B+ tree with two components, the classical component and the quantum component. The classical component follows the classical B+ tree, which allows us to leverage its effective balanced tree structure. The quantum component stores a concise "replication" of the hierarchical relationships in the B+ tree in the quantum memory, which could load the relationships in the form of superposition efficiently due to quantum parallelism.

Empowered by the two-component design of our quantum B+ tree, we propose a hybrid quantum-classical algorithm called the *Global-Classical Local-Quantum* (GCLQ) search to solve the quantum range query problem. It involves two main steps. The first step is called the *global classical search*, which finds at most two candidate nodes from the classical B+ tree. It is guaranteed that all the relevant results are covered in the candidate nodes and account for a significant amount under the candidate nodes (i.e., at least $\frac{1}{8B}$ of the items under candidate nodes are the relevant results). Meanwhile, the global classical search is very efficient owing to the effective structure of B+ tree. The second step is called the *local quantum search*, which returns a superposition of all the exact query results from the candidate nodes with efficient quantum parallelism techniques in the quantum memory. As a result, the time complexity of our proposed GCLQ search is $O(C_l \log N)$, where C_l denotes the cost of a load operation on the quantum memory, which is asymptotically optimal in quantum computers. Based on previous studies [31, 34, 36, 39–42], C_l is commonly regarded as a constant since the quantum memory is believed to support very efficient load and store operations in future quantum computers, and thus the complexity can be simplified to be $O(\log N)$. This improves the optimal classical result by reducing the $O(k)$ cost and is much more efficient than the $O(\sqrt{Nk})$ time complexity of the existing quantum algorithms without using any data structure.

We also extend our quantum B+ tree to solve the dynamic quantum range query. Our two-component design allows us to flexibly extend our quantum B+ tree to the B+ tree variants. As such, we extend our quantum B+ tree to the *dynamic quantum B+ tree* by adapting the logarithmic method [11], which supports inserting a new item into the tree and deleting an existing item from the tree. All the insertions and deletions are also replicated to the quantum component of the tree, which is efficient due to the conciseness of the quantum component. We show that each insertion and deletion can be done in $O(C_s \log N)$ time and $O((C_l + C_s) \log N)$, where C_s denotes the cost of a store operation on the quantum memory (and thus both of the insertion and deletion complexity can be simplified as $O(\log N)$), and the dynamic range query can be done in $O(C_l \log^2 N)$ (which can be simplified as $O(\log^2 N)$) with the dynamic quantum B+ tree.

In summary, our contributions are shown as follows.

- We are the first to study the quantum range query problems.
- We are the first to propose a tree-like quantum data structure, which is the quantum B+ tree.
- We design a hybrid quantum-classical algorithm that can answer a quantum range query on a quantum B+ tree in $O(C_l \log N)$ ($\approx O(\log N)$) time, which does not depend on the output size and is asymptotically optimal in quantum computers.

- We further extend the quantum B+ tree to the dynamic quantum B+ tree that supports insertions and deletions in $O(C_s \log N)$ ($\approx O(\log N)$) and $O((C_l+C_s)\log N)$ ($\approx O(\log N)$) time, respectively, and the complexity of the quantum range query on the dynamic quantum B+ tree is $O(C_l \log^2 N)$ ($\approx O(\log^2 N)$).
- We conducted experiments to confirm the significant speedup of the quantum range queries on real-world datasets. In our experiments, our quantum data structure is up to 1000× more efficient in IO cost than the classical data structure and 3.3x faster in the estimated execution time.

The rest of the paper is organized as follows. In Section 2, we first introduce some basic knowledge about quantum algorithms. We formally define the quantum range query problems in Section 3. In Section 4, we show the design of the quantum B+ tree and introduce our algorithm to answer the quantum range query. In Section 5, we extend the quantum B+ tree to the dynamic version. Section 6 presents our experimental studies. In Sections 7 and 8, we introduce the related work and conclude our paper, respectively.

2 PRELIMINARIES

A *bit* is a basic unit when we store data in the memory or on disks. In classical computers, a bit has two *states* (i.e., 0 and 1). A bit in quantum computers is known as a *quantum bit*, which is called a *qubit* in this paper for simplicity. Similar to a bit in classical computers, a qubit also has states. Following [9, 20], we introduce the "Dirac" notation (i.e., "$|\cdot\rangle$") to describe the states of a qubit. Specifically, we write an integer in $|\cdot\rangle$ (e.g., $|0\rangle$) to describe a *basis state*, and we write a symbol in $|\cdot\rangle$ (e.g., $|q\rangle$) to describe a *mixed state* (more formally known as a *superposition*). Note that the symbol in a superposition of a qubit is used to denote this qubit.

We have two basis states of a qubit, $|0\rangle$ and $|1\rangle$, which correspond to state 0 and state 1 of a classical bit, respectively. Different from a classical bit, a qubit q could have a state "between" $|0\rangle$ and $|1\rangle$, (described by a superposition $|q\rangle$). Specifically, superposition $|q\rangle$ is represented as a linear combination of the two basis states: $|q\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are two complex numbers called the *amplitudes* and we have $|\alpha|^2 + |\beta|^2 = 1$. In quantum computers, instead of directly obtaining the value of a bit, we *measure* a qubit q, which will "collapse" its superposition $|q\rangle$ and obtain the result state 0 (resp. 1) with probability $|\alpha|^2$ (resp. $|\beta|^2$). For instance, let q be a qubit with superposition $|q\rangle = (0.36+0.48i)|0\rangle + (0.48-0.64i)|1\rangle$. If we measure q, we can obtain 0 with probability $|0.36+0.48i|^2 = 0.36$ or 1 with probability $|0.48 - 0.64i|^2 = 0.64$.

Similar to a register in a classical computer, we define a *quantum register* to be a collection of qubits. Consider a quantum register with two qubits. The basis states of the two qubits will contain all the combinations of the basis states of single qubits, i.e., $|0\rangle|0\rangle$, $|0\rangle|1\rangle$, $|1\rangle|0\rangle$ and $|1\rangle|1\rangle$. If we measure one of the qubits, this measurement may change the state of the other qubit. Such phenomenon is widely known as *quantum entanglement* [52].

Consider the following example. For simple illustration, all the amplitudes in this example only have real parts (of complex numbers). Let us assume ψ to be a quantum register with two qubits q_0 and q_1, where the measurement of q_0 will change the state of q_1. Specifically, $|q_1\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, which is measured to be 0 or 1 with probability both equal to $\frac{1}{2}$. If 0 is obtained, the state of q_0 will be changed to $|q_0\rangle = 0.6|0\rangle + 0.8|1\rangle$, and otherwise, it will be changed to $|q_0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. As such, the state of q_0q_1 (i.e., ψ) can be represented as

$$|\psi\rangle = |q_1\rangle|q_0\rangle = \frac{1}{\sqrt{2}}|0\rangle(0.6|0\rangle + 0.8|1\rangle) + \frac{1}{\sqrt{2}}|1\rangle(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle)$$
$$= \frac{0.6}{\sqrt{2}}|0\rangle|0\rangle + \frac{0.8}{\sqrt{2}}|0\rangle|1\rangle + \frac{1}{2}|1\rangle|0\rangle + \frac{1}{2}|1\rangle|1\rangle.$$

It indicates that the two *entangled* qubits have 4 basis states $|0\rangle|0\rangle$, $|0\rangle|1\rangle$, $|1\rangle|0\rangle$ and $|1\rangle|1\rangle$ with amplitudes $\frac{0.6}{\sqrt{2}}$, $\frac{0.8}{\sqrt{2}}$, $\frac{1}{2}$ and $\frac{1}{2}$, respectively. We can extend this system to a quantum register with n qubits which have 2^n basis states and the corresponding 2^n amplitudes.

The states of qubits can be transformed by *quantum gates* in a *quantum circuit*, which work similarly as the gates and circuits in classical computers. We can also measure a qubit in a quantum circuit. In the following, we introduce three common types of quantum gates that will be used in this paper, namely a *Hadamard gate*, an *X-gate* and a *controlled-X gate*. A *Hadamard gate* [29, 43] transforms $|0\rangle$ into $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and transforms $|1\rangle$ into $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. An *X-gate* [43] swaps the amplitudes of $|0\rangle$ and $|1\rangle$. A *controlled-X* gate has two parts, the *control* qubit, says q_1, and the *target* qubit, says q_0. A controlled-X gate applies an X-gate on the target qubit q_0 if the control qubit is $|1\rangle$, and otherwise, it keeps the target qubit unchanged.

For example, assume an input $|q\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. After being transformed by a Hadamard gate, we have $|q\rangle = \frac{1}{\sqrt{2}}(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle) + \frac{1}{\sqrt{2}}(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle) = |0\rangle$. Clearly, if we now measure q, we will always obtain 0. We give more examples about the X-gate and the controlled-X gate shortly.

A quantum circuit could involve the combination of multiple input qubits and multiple gates. Since the state transformations and measurements for each qubit follow the timeline of a wire, we could describe the state changes in an ordered list of events. For example, we are given a quantum circuit with two input qubits q_0 and q_1 that is described as follows.

(1) Apply an X-gate on q_1.
(2) Apply a controlled-X gate where the control qubit is q_1 and the target qubit is q_0.

Consider the following three examples of input:

(1) $|q_1\rangle|q_0\rangle = |0\rangle|0\rangle$. After the X-gate on q_1, $|q_1\rangle|q_0\rangle = |1\rangle|0\rangle$. Since $|q_1\rangle = |1\rangle$, we apply an X-gate on q_0, and thus the result is $|1\rangle|1\rangle$.
(2) $|q_1\rangle|q_0\rangle = |1\rangle|1\rangle$. After the X-gate on q_1, $|q_1\rangle|q_0\rangle = |0\rangle|1\rangle$. Since $|q_1\rangle = |0\rangle$, we do nothing, and thus the result is $|0\rangle|1\rangle$.
(3) $|q_1\rangle|q_0\rangle = \frac{1}{\sqrt{2}}|0\rangle|0\rangle + \frac{1}{\sqrt{2}}|1\rangle|1\rangle$. After the X-gate on q_1, $|q_1\rangle|q_0\rangle = \frac{1}{\sqrt{2}}|1\rangle|0\rangle + \frac{1}{\sqrt{2}}|0\rangle|1\rangle$. The result is $\frac{1}{\sqrt{2}}|1\rangle|1\rangle + \frac{1}{\sqrt{2}}|0\rangle|1\rangle$.

Following prior studies [26, 54, 58, 63], we call such a quantum transformation consisting of a series of quantum gates in a quantum circuit as a *quantum oracle*. A quantum oracle can be regarded as a black box of a quantum circuit where we only focus on the quantum operation it can perform but skip the detailed circuit. Since different gate sets (which correspond to the instruction sets in classical computers) may cause different time complexities and

the general quantum computer is still at a very early stage, we normally use the query complexity to study quantum algorithms, which is to measure the number of queries to the quantum oracles.

3 PROBLEM DEFINITION

In this section, we formally define the quantum range query problems in the *static* and *dynamic* cases, respectively.

We first consider the *static* quantum range query problem (for simplicity, it is simply called the quantum range query problem). We are given an *immutable* dataset D of N items. Each item in D is represented as a key-record pair (key_i, rec_i) for $i \in [0, N-1]$. Note that the subscription i in (key_i, rec_i) represents the *index* of this pair, and the indices start from 0. Each key key_i is an integer, and each record rec_i is a bit-string. Note that in our problem, the keys are integers, but they can be easily extended to float numbers. A range query of lower bound x and upper bound y is denoted as $QUERY(x, y)$. In classical range query problem, $QUERY(x, y)$ returns a list of key-record pairs whose keys fall in range $[x, y]$. Let k be the number of items that $QUERY(x, y)$ returns, and let $l_0, l_1, \ldots, l_{k-1}$ be the indices of the returned items. The returned list of $QUERY(x, y)$ is thus represented as $\{(key_{l_0}, rec_{l_0}), \cdots, (key_{l_{k-1}}, rec_{l_{k-1}})\}$. In the quantum range query problem, we aim to return a superposition of all the key-record pairs in the list. We formally define the quantum range query problem as follows.

Definition 3.1 (Quantum Range Query). Given an immutable dataset $D = \{(key_0, rec_0), \ldots, (key_{N-1}, rec_{N-1})\}$ and two integers x and y where $x \le y$, a quantum range query $QUERY(x, y)$ is to return the following superposition

$$\frac{1}{\sqrt{k}} \sum_{i=0}^{k-1} \left|key_{l_i}\right\rangle \left|rec_{l_i}\right\rangle,$$

such that for each $i \in [0, k-1]$, $x \le key_{l_i} \le y$.

Note that in the quantum range query, we return a superposition which is the linear combination of all the desired key-record pairs. It can be observed that for each $i \in [0, k-1]$, the probability of obtaining $\left|key_{l_i}\right\rangle \left|rec_{l_i}\right\rangle$ is equal to $\frac{1}{k}$.

Next, we also define the *dynamic* quantum range query. In the dynamic case, instead of having an immutable dataset, we consider a *dynamic* dataset D that supports the *insertion* and *deletion* operations. Specifically, an insertion operation inserts a new item (key, rec) into D and returns the new dataset $D \cup (key, rec)$. A deletion operation delete an existing item (key, rec) from D and returns the new dataset $D \setminus (key, rec)$. The dynamic quantum range query is formally defined as follows.

Definition 3.2 (Dynamic Quantum Range Query). Given a dataset $D = \{(key_0, rec_0), \ldots, (key_{N-1}, rec_{N-1})\}$ that supports the insertion and deletion operations and two integers x and y where $x \le y$, a dynamic quantum range query $QUERY(x, y)$ is to return the following superposition

$$\frac{1}{\sqrt{k}} \sum_{i=0}^{k-1} \left|key_{l_i}\right\rangle \left|rec_{l_i}\right\rangle,$$

such that for each $i \in [0, k-1]$, $x \le key_{l_i} \le y$.

Note that the only difference between the dynamic and the static quantum range query is whether the given dataset is dynamic.

In our quantum range query problems, we aim to return a superposition of all the desired results, which is new in the research area to the best of our knowledge. Although quantum range queries cannot return a classical list of all answers, we still consider it very meaningful, since we do not regard the result as a final output available to a user but as input to other quantum algorithms, such as quantum database queries [16], the quantum recommendation system [35] and quantum machine learning [2], as mentioned in Section 1. Consider a movie database as an example. We want to find the movie ranked the highest that was released between 1990 and 1999. A common efficient query execution plan would be first retrieving all the k movies released between 1990 and 1999 as the candidates (which is a range query), and then finding the one among them with the highest ranking. For the latter step, traditionally, a cost of $O(k)$ is required to list out and traverse all the k candidates for finding the query result, which could be time-consuming if k is large. When applying our quantum range query, the k candidates in the form of superposition can be used as input to some efficient quantum database search algorithms [3, 16] that could find the highest-ranking movie in $O(\sqrt{k})$ time, which is much more efficient.

4 QUANTUM B+ TREE

In this section, we propose the quantum B+ tree to solve the quantum range query problems. To the best of our knowledge, no prior studies explored using a data structure in quantum computers. Compared with performing a quantum algorithm on *unstructured* dataset, using quantum data structures could answer our quantum range query problems more efficiently. We focus on the static quantum B+ tree in this section, and discuss the dynamic variant later in Section 5.

Before introducing the details of the quantum B+ tree, we first introduce the concept of *Quantum Random Access Memory* (QRAM) which is used to store the quantum data structure and give query results in superpositions. Like the classical RAM, we have QRAM in quantum computers to read and write quantum states. Following [35, 50], we assume the existence of a *classical-write quantum-read QRAM*. It can store a classical value into a given address, and it can accept a superposition of *multiple* addresses and returns a superposition of the corresponding values due to quantum parallelism. Formally, we define QRAM as follows.

Definition 4.1 (Quantum Random Access Memory (QRAM)). A QRAM Q is an ideal model which can perform the following store and load operations.

- A store operation denoted by $Q[addr] = val$ stores value val into address $addr$, where $addr$ and val are two bit-strings.
- A load operation denoted by

$$Q \frac{1}{\sqrt{k}} \sum_{i=0}^{k-1} |addr_i\rangle |a\rangle = \frac{1}{\sqrt{k}} \sum_{i=0}^{k-1} |addr_i\rangle |a \oplus val_i\rangle \tag{1}$$

  takes a superposition of a number k of addresses (i.e., $addr_i$ for $i \in [0, k-1]$) as input and loads a superposition of the values of these addresses into the destination quantum register of initial state a where $\oplus$ denotes the XOR operation.

In our definition of QRAM, the concept of store operation is similar to a classical RAM where we store val in a memory block

at address $addr$. For simplicity, we also use $Q[addr]$ to denote the value stored at address $addr$. In the load operation, the input of multiple addresses and the output of multiple values are handled in superpositions, which is the major difference from a classical RAM. The result of a load operation is written to a quantum register using the XOR operation (i.e., $\oplus$). For instance, if we set the initial state of the destination quantum register to be $|0\rangle$, then the exact values will be loaded as follows.

$$Q\frac{1}{\sqrt{k}}\sum_{i=0}^{k-1}|addr_i\rangle|0\rangle = \frac{1}{\sqrt{k}}\sum_{i=0}^{k-1}|addr_i\rangle|Q[addr_i]\rangle \quad (2)$$

Quantum computer involving many qubits is still under development and thus, QRAM is still under development. Thus, in the following, we use C_s and C_l to denote the cost of a store operation and a load operation on QRAM, respectively.

With the concept of QRAM, we can form the following method of answering the quantum range query problems in two steps using the existing quantum algorithms on the *unstructured* dataset. In the first step of pre-processing (for loading the dataset), for each key-record pair (key_i, rec_i), we store it into a QRAM Q using the store operation $Q[key_i] = rec_i$. Clearly, the total time complexity is $O(N)$. In the second step of data retrieval, we first find all the addresses in Q that are within the query range, which costs $O(\sqrt{Nk})$ time using the state-of-the-art quantum algorithm [14, 15], and then use the load operation to find the resulting superposition of the corresponding values. Totally, the data retrieval step costs $O(\sqrt{Nk})$. When N grows large, this is even worse than the classical range query algorithms (e.g., using the B+ tree) with time complexity logarithm to N.

To solve the quantum range query problems more efficiently with QRAM, we propose our quantum data structure called the quantum B+ tree. The major design of our quantum B+ tree is derived from the basic idea of the classical B+ tree. It is well known that the classical B+ tree utilizes a balanced tree structure, which performs a range query in the dataset D of size N efficiently with time complexity $O(\log N + k)$, where k is the return size. In classical computers, this time complexity is shown to be asymptotically optimal. For the quantum range queries in quantum computers, we are also interested in the best possible time complexity. To explore that, we introduce another problem called the *membership* problem, which, given a query key, decides whether this key exists in the dataset D. Existing studies have shown that the time complexity lower bound of solving the membership problem in quantum computers is $\Omega(\log N)$ [5, 53]. Based on that, we present the following lemma to show the time complexity lower bound of solving our quantum range queries. For the sake of space, we give a proof sketch for each lemma or theorem in this paper, and the full proof can be found in our technical report [8].

Lemma 4.2. *The time complexity to answer a quantum range query is $\Omega(C_l \log N)$, where C_l is the cost of a load operation on QRAM.*

Proof sketch. Consider a quantum range query $QUERY(x, x)$. It answers whether the key x exists in the dataset, which is a membership problem of time complexity $\Omega(C_l \log N)$ by [53]. Therefore, the time complexity of answering a quantum range query is also $\Omega(C_l \log N)$. □

Note that based on [31, 34, 36, 39–42], C_l is commonly regarded as a constant, and thus $\Omega(C_l \log N)$ could be simplified as $\Omega(\log N)$.

According to the above lemma, we cannot achieve a better time complexity than $\Omega(\log N)$ for the quantum range query. The goal of our static quantum B+ tree is thus to obtain this asymptotically optimal time complexity $\Omega(\log N)$. Since the $O(k)$ time cost of listing all the k returned results cannot be reduced in classical range queries, we consider eliminating this cost in our quantum B+ tree, which is feasible in quantum computers that handle the superpositions due to quantum parallelism.

Based on the above analysis, in the following, we first introduce the details of how we design our quantum B+ tree in Section 4.1, and then, we propose our quantum range search algorithm called the *global-classical local-quantum search* in Section 4.2.

4.1 Design of Quantum B+ Tree

The design of our quantum B+ tree includes the following two core ideas. Firstly, a quantum B+ tree consists of the classical component and the quantum component. The classical component is the same as the classical B+ tree, and the quantum component stores a concise replication of the hierarchical relationships and node data in the classical component. This is to take advantage of the balanced tree structure of the classical B+ tree, which has a tree *height* of $O(\log N)$ and thus retains the $O(\log N)$ search efficiency. Secondly, the quantum component utilizes QRAM to process the tree structure and return the range search results in superpositions efficiently by quantum parallelism. Note that the classical component can be stored in either a classical computer or a quantum computer, since a classical system can be regarded as a quantum system using only $|0\rangle$ and $|1\rangle$ without any superposition. For consistency, we store both the classical and the quantum components into the quantum system, so that all operations in our proposed data structures and algorithms are in the quantum computers.

In the following, we first introduce some basic concepts of the classical B+ tree, which is the prototype of our quantum B+ tree. We consider a *weight-balanced B+ tree* [6]. It is a tree data structure that consists of nodes in two types, the internal nodes and the leaves. Each node has a unique ID in the form of an integer, which is used to represent this node. A user parameter B called the branching factor is set to be an integer that is a power of 2 and is at least 4. A leaf contains B key-record pairs sorted by the keys in ascending order. An internal node has B children, each of which is a leaf or an internal node. Each node is also associated with a routing key, says (L, U), which represents a range with lower bound L and upper bound U, and all the key-record pairs under the internal node with routing key (L, U) have keys within this range. The children under an internal node have non-overlapping routing keys and are sorted by the lower bound of their routing keys. A node or a key-record pair under a leaf could be empty, which is represented as a special mark called *dummy*. We assume that *dummy* appears only in the last positions under a node.

The left part in Figure 1 illustrates an example of the classical component of a classical B+ tree with $N = 14$ key-record pairs and the branching factor $B = 4$. It has four internal nodes of ID from 0 to 3 and seven leaves of ID from 4 to 10. For instance, the node of ID 1 (simply called node 1) has routing key (1, 6), indicating that

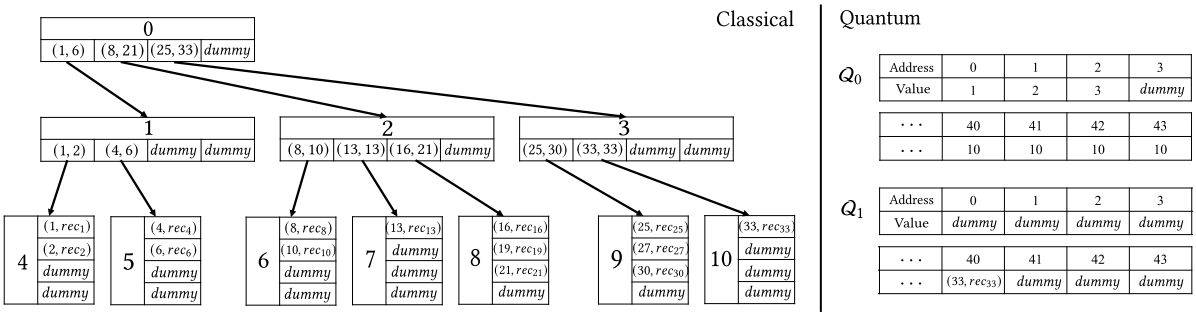


Figure 1: An Example of a Quantum B+ Tree where $N = 14$ and $B = 4$

all the data items with keys in range $[1, 6]$ are under node 1. Also, node 1 has four children, two of which are non-dummy leaves and the other two are dummy nodes. For the leave shown as node 4 with routing key (1, 2), two data items (with keys equal to 1 and 2) and two dummy items are assigned.

We define the *level* of a node to be its distance to the root, where the distance between two nodes is defined to be the minimum number of edges connecting these two nodes. In a B+ tree, all the leaves have the same level. We also define the *height* of a node to be its distance to a leaf, and the height of the B+ tree is defined to be the height of the root. We define the *weight* of a node to be the number of all non-dummy key-record pairs under this node. A non-root node of height h is said to be *balanced* if its weight is between $\frac{1}{4}B^{h+1}$ and B^{h+1}. Further, a non-root node of height h is said to be *perfectly balanced* if its weight is between $\frac{1}{2}B^{h+1}$ and B^{h+1}.

In our running example shown in the left part of Figure 1, there are 4 non-dummy key-record pairs under node 1 of height h equal to 1, and thus the weight of node 1 is equal to 4. This indicates that node 1 is balanced (since 4 is no less than $\frac{1}{4}B^{h+1} = 4$) but is not perfectly balanced (since 4 is less than $\frac{1}{2}B^{h+1} = 8$).

A B+ tree is said to be a weight-balanced B+ tree if all its non-root nodes are balanced and its root has at least two non-dummy children. It is easy to verify that the height of a weight-balanced B+ tree with N key-records pairs is $O(\log N)$. Moreover, a weight-balanced B+ tree is said to be perfectly balanced if all its non-dummy nodes are perfectly balanced. It can be verified that the B+ tree shown in the left part of Figure 1 is a weight-balanced B+ tree but is not perfectly balanced.

After introducing the structure of a classical B+ tree, we now discuss how we represent the hierarchical relationships and node data of a B+ tree in QRAM. Since each node contains B elements (including dummy), each of which has a child node with a routine key or a key-record pair, we could fit a classical B+ tree into a QRAM storing $M * B$ key-value pairs, where M denotes the total number of nodes in the classical B+ tree, a key represents the ID of a tree node and a value stores the related information of this node. This can be done by a traversal of all the tree nodes in any order (in this paper, we choose the breadth-first order for simplicity).

We use two QRAMs, namely the hierarchy QRAM Q_0 and the data QRAM Q_1, to store the hierarchical relationships (i.e., the mapping from each node to its children) and the data (i.e., the routine key of a node or the key-record pair of a leave), respectively. Specifically, we perform $M * B$ store operations on Q_0 where M denotes the total number of nodes in the classical B+ tree as follows. Let $C_{i,j}$ (for $i \in [0, M-1]$ and $j \in [0, B-1]$) denote the node ID of the j-th child of node i in the B+ tree. Specially, if node i is a leave with no children, we set $C_{i,j} = i$. Also, if the j-th child of node i is *dummy*, we assign *dummy* to $C_{i,j}$. Then, for each node $i \in [0, M-1]$ and each $j \in [0, B-1]$, we perform a store operation to store value $C_{i,j}$ at address $i * B + j$, i.e., $Q_0[i * B + j] = C_{i,j}$.

In the right part of Figure 1, we show an example of the quantum B+ tree. In the hierarchy QRAM Q_0, totally 44 (= $11 * 4$) addresses are "occupied". Among them, for instance, node 0 involves storing the IDs of its three non-dummy children (i.e., 1–3) and one *dummy* into addresses 0–3, respectively, and node 10 involves storing value 10 for all its corresponding addresses 40–43 since it is a leaf.

The data QRAM Q_1 stores the routine key of each node or the key-record pair of each leave. Similar to Q_0, we also perform $M * B$ store operations on Q_1. Consider node i in the B+ tree. If i is an internal node, then for its j-th child, we assign the routine key (i.e., a 2-tuple of lower bound and upper bound of this routine key) to a variable, says $P_{i,j}$, which is *dummy* if the j-th child is dummy. If i is a leaf, we set $P_{i,j}$ to be the j-th key-record pair of node i or *dummy* if the j-th key-record pair is dummy. As such, for each node $i \in [0, M-1]$ and each $j \in [0, B-1]$, we perform a store operation to store pair $P_{i,j}$ at address $i * B + j$, i.e., $Q_1[i * B + j] = P_{i,j}$.

Continuing our running example in Figure 1, in the data QRAM Q_1, we also store values into 44 addresses similarly, where for node 0, since it is an internal node, all values stored are *dummy* (at addresses 0–3), and for node 10 with two key-record pairs and two dummies, the values at address 40–43 are stored correspondingly.

Note that both QRAMs have data across the same number (i.e., $M * B$) of addresses, and at the same address, the value stored are related to the same node or the same key-record pair in a leaf. Thus, it is easy to combine the two QRAMs into one QRAM by using a larger quantum register storing the values of both QRAMs at the same address for implementation. In this paper, for clear structuring and illustration, we use two QRAMs to separate them.

For both Q_0 and Q_1, we could easily retrieve multiple values for the relationships in the tree with one load operation on QRAM. Specifically, in the node QRAM Q_0, we perform the following load

operation for node i ($i \in [0, M-1]$)

$$\begin{aligned} &Q_0 \frac{1}{\sqrt{B}} \sum_{j=0}^{B-1} |i*B+j\rangle |0\rangle = \frac{1}{\sqrt{B}} \sum_{j=0}^{B-1} |i*B+j\rangle \left|C_{i,j}\right\rangle \\ &= \frac{1}{\sqrt{B}} \sum_{j=0}^{f_i-1} |i*B+j\rangle \left|C_{i,j}\right\rangle + \frac{1}{\sqrt{B}} \sum_{j=f_i}^{B-1} |i*B+j\rangle |dummy\rangle \end{aligned} \tag{3}$$

where f_i denotes the number of non-dummy children among all the children of node i. This operation loads the IDs of all the children of node i into a superposition in C_l time. Similarly, on Q_1, we can load the key-record pairs under a leaf (with node ID i) with the following load operation

$$\begin{aligned} &Q_1 \frac{1}{\sqrt{B}} \sum_{j=0}^{B-1} |i*B+j\rangle |0\rangle = \frac{1}{\sqrt{B}} \sum_{j=0}^{B-1} |i*B+j\rangle \left|P_{i,j}\right\rangle \\ &= \frac{1}{\sqrt{B}} \sum_{j=0}^{f_i-1} |i*B+j\rangle \left|P_{i,j}\right\rangle + \frac{1}{\sqrt{B}} \sum_{j=f_i}^{B-1} |i*B+j\rangle |dummy\rangle . \end{aligned} \tag{4}$$

4.2 Global-Local Quantum Range Search

In this section, we propose an algorithm called the *global-local quantum range search* to solve the quantum range query with our quantum B+ tree.

Recall that our quantum range query aims to return a superposition of all the key-record pairs such that each key in the result is within a query range $[x, y]$. To better present our algorithm, we first propose a technique called *post-selection* which can answer the quantum range query with our dataset stored in a QRAM even in an *unstructured* way (i.e., without using the quantum B+ tree).

The major idea of post-selection is based on the efficient processing of superpositions of QRAM and the quantum oracle. Specifically, we first apply the store operations to store all the key-record pairs in the dataset D in a QRAM, says Q', which is a *pre-processing* step. That is, for each $i \in [0, N-1]$, we do a store operation $Q'[i] = (key_i, rec_i)$. We can thus obtain a superposition of all the key-record pairs as $\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |key_i\rangle |rec_i\rangle$ by a load operation on Q' in C_l time. After obtaining this superposition, we design a quantum oracle as follows to *post-select* the superposition of the desired items within the query range $[x, y]$.

Let *in* denote the set of indices of all the key-record pairs in the dataset such that the keys are within $[x, y]$, and let *out* denote the set of all the remaining indices. First, we place an additional qubit with initial state $|0\rangle$ in the end for the purpose of controlling and measuring. Thus, we obtain $\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |key_i\rangle |rec_i\rangle |0\rangle$. Next, we perform a quantum oracle which transforms the above superposition based on a controlled-X gate controlled by whether key_i is within $[x, y]$. Specifically, if $x \le key_i \le y$, the last qubit will be transformed to $|1\rangle$, and otherwise, no transformation is performed. After the transformation, we obtain $\frac{1}{\sqrt{N}} \sum_{i \in in} |key_i\rangle |rec_i\rangle |1\rangle + \frac{1}{\sqrt{N}} \sum_{i \in out} |key_i\rangle |rec_i\rangle |0\rangle$. Note that k (resp. $N-k$) denotes the size of *in* (resp. *out*). The above state can be rewritten as $\frac{\sqrt{k}}{\sqrt{N}} \cdot \frac{1}{\sqrt{k}} \sum_{i \in in} |key_i\rangle |rec_i\rangle |1\rangle + \frac{\sqrt{N-k}}{\sqrt{N}} \cdot \frac{1}{\sqrt{N-k}} \sum_{i \in out} |key_i\rangle |rec_i\rangle |0\rangle$, where the part $\frac{1}{\sqrt{k}} \sum_{i \in in} |key_i\rangle |rec_i\rangle$ is exactly the quantum range query result we desire. Therefore, finally, we measure the last qubit, which will obtain the desired superposition of all the key-record pairs in set *in* (i.e., $\frac{1}{\sqrt{k}} \sum_{i \in in} |key_i\rangle |rec_i\rangle$) with probability equal to $\frac{k}{N}$ (i.e., the last qubit is measured to be 1). Since our desired result could not be deterministically obtained, we repeat the final measurement step until the last qubit is measured to be 1. It is easy to verify that N/k repeats are needed in expectation.

Overall, to answer the quantum range query, the online processing of post-selection has expected time complexity at least $O(N/k)$, since the final step is repeated N/k times in expectation. In range queries, the number of results k is normally much smaller than the dataset size N, and thus the $O(N/k)$ time cost could be too large. Therefore, the post-selection technique as a standalone approach is inefficient, since the dataset is organized in an unstructured manner in QRAM.

However, if k and N are approximately equal, the cost of $O(N/k)$ could be small (e.g., close to an $O(1)$ cost) for a post-selection. Motivated by this, in the following, we propose our efficient quantum range query algorithm based on our quantum B+ tree, which leverages post-selection as a sub-step.

Our quantum range query algorithm is called the *Global-Classical Local-Quantum search* (GCLQ search). It has two major steps. The first step is the *global classical* search, which aims to find the candidate B+ tree nodes *precisely* from the classical B+ tree such that they contain all the query results but the number of irrelevant results is as small as possible. The second step is the *local quantum* search, which searches from the QRAMs in the quantum B+ tree and returns the result in superposition efficiently with a post-selection step. In the following two subsection, we introduce the two steps, respectively.

4.2.1 Global Classical Search. We first introduce the global classical search step. In this step, we traverse the classical B+ nodes from the root to obtain the "precise" candidate nodes that are without too many irrelevant results. Although the classical B+ is already capable of returning all the exact results, we do not expect returning many detailed-level candidate nodes in this step (which could cause the undesired cost of listing all the candidate nodes). Instead, we postpone the processing of detailed-level nodes for fast local search in the quantum component, and we only return at most *two* candidate nodes which are as "precise" as possible.

Specifically, given a B+ tree node with routing key (L, U) and the query range $[x, y]$, we classify this node into three categories, namely an *outside* node if range $[L, U]$ and range $[x, y]$ have no overlap, a *partial* node if $[L, U]$ is partially inside $[x, y]$, and an *inside* node if $[L, U]$ is completely inside $[x, y]$. A partial node i is said to be *precise* if i is a leaf or there exists an inside node among the children of node i. It can be verified that if a partial node is precise and the B+ tree is weight-balanced, then a sufficiently significant portion of key-record pairs under this node are included in the query range $[x, y]$ (which is formalized in a lemma shortly).

Now, we present the detail of our global classical search, which aims to return precise partial nodes with level as small as possible (to avoid processing detailed-level nodes). For simplicity, we assume that the root node is a partial node (otherwise it falls to special cases where the results involve no item or all items). We create an empty list $\mathcal{L}$ of tree nodes and add the root to the list initially. Then, we loop the following steps until $\mathcal{L}$ is empty. (1) We check whether

there exists a precise node in $\mathcal{L}$. (2) If the answer of Step (1) is yes, we immediately add all nodes in $\mathcal{L}$ into the returned candidate set and terminate. Otherwise, we replace each node in $\mathcal{L}$ with all its non-outside children (since outside nodes cannot contain the desired results).

For example, consider a query $QUERY(5, 11)$ given to the quantum B+ tree in Figure 1. We first check the root (i.e., node 0) with three children. We find that two of them (i.e., node 1 and 2) are partial nodes and one of them (i.e., node 3) is an outside node. Thus, the root is not precise (i.e., there does not exist a precise node in $\mathcal{L}$), and we replace the root with node 1 and node 2 in $\mathcal{L}$. Then, we check whether node 1 or node 2 is precise. We find that node 2 is precise since it contains an inside child (i.e., node 6). Therefore, we obtain the returned candidate set containing node 1 and node 2.

The following lemma shows the effectiveness of the global classical search.

Lemma 4.3. *Given a query range $[x, y]$ and a B+ tree that is weight-balanced, the returned candidate set of the global classical search contains at most two nodes. Moreover, let $\mathcal{R}$ be the set of all key-record pairs under the above returned candidate nodes, and let $\mathcal{R}^*$ be the set of all key-record pairs such that the keys are within the query range $[x, y]$. Then, $\mathcal{R}^* \subset \mathcal{R}$ and $|\mathcal{R}^*| \geq \frac{1}{8B}|\mathcal{R}|$.*

Proof sketch. Since the routing keys of all nodes in the same level are disjoint, we cannot have more than two partial nodes in the same level. It is also easy to verify that the returned nodes are from the same level, Thus, the candidate set contains at most two nodes. Since we only filter out the outside node in this algorithm, we have $\mathcal{R}^* \subset \mathcal{R}$. Finally, since the precise partial node either is a leaf (which contains at least $1/B$ items in the query range), or contains an inside child (which contains at least $\frac{1}{4B}$ items in the query range due to the balanced nodes of the B+ tree), and it can be verify that at least one returned node is precise, we have $|\mathcal{R}^*| \geq \frac{1}{8B}|\mathcal{R}|$. □

4.2.2 Local Quantum Search. Now, we introduce the local quantum search. Since the local quantum search starts from at most two candidate nodes, consider answering $QUERY(x, y)$ starting from a node u and another node v as the candidate nodes in level j and the height of the tree is h. Step 1 is to initialize the first n_i quantum qubits to be $\frac{1}{\sqrt{2}}(|u\rangle + |v\rangle)$, where n_i denotes the number of bits to store each node index i. Then, in Step 2, we add $n_b = \log_2 B$ auxiliary qubits $|0\rangle$ to the last and also apply a Hadamard gate on each auxiliary qubit. We obtain

$$\frac{1}{\sqrt{2}}(|u\rangle + |v\rangle)\mathcal{H}|0\rangle\mathcal{H}|0\rangle\cdots\mathcal{H}|0\rangle = \frac{1}{\sqrt{2B}}(|u\rangle + |v\rangle)\sum_{i=0}^{B-1}|i\rangle.$$

This step is to enumerate all the edges from the candidate nodes. Then, in Step 3, we add n_i qubits to the end and apply Q_0 to obtain all the children of the candidate nodes, so we obtain

$$\frac{1}{\sqrt{2B}}|u\rangle\sum_{i=0}^{B-1}|i\rangle|c_i\rangle + \frac{1}{\sqrt{2B}}|v\rangle\sum_{i=0}^{B-1}|i\rangle|c_{i+B}\rangle,$$

where $c_0, \cdots, c_{B-1}$ are the B children of u and $c_B, \cdots, c_{2B-1}$ are the B children of v. If we only look at the last n qubits, we obtain $\frac{1}{\sqrt{2B}}\sum_{i=0}^{2B-1}|c_i\rangle$, which is the $2B$ children of u and v. We repeat Step 2 and Step 3 for $h - j$ times so that we obtain all the $2B^{h-j}$ leaves below u and v. Then, we do the same thing as Step 2 to enumerate all the $2B^{h-j+1}$ key-record pairs in the $2B^{h-j}$ leaves. In the last step, we apply Q_1 to obtain all the key-record pairs below u and v and then do a post-selection search. Denote the $2B^{h-j+1}$ key-record pairs as $(key_0, rec_0), \cdots, (key_{2B^{h-j+1}-1}, rec_{2B^{h-j+1}-1})$. Then the quantum state becomes $\frac{1}{\sqrt{2B^{h-j+1}}}\sum_{i=0}^{2B^{h-j+1}-1}|key_i\rangle|rec_i\rangle$. We also use $|in\rangle$ to denote the k key-record pairs in the query range and use $|out\rangle$ to denote the other dummy key-record pairs and non-dummy key-record pairs which are not in the query range. We obtain $\frac{\sqrt{k}}{\sqrt{2B^{h-j+1}}}|in\rangle + \frac{\sqrt{2B^{h-j+1}-k}}{\sqrt{2B^{h-j+1}}}|out\rangle$. If we do a post-selection, we can obtain $|in\rangle$ with probability $\frac{k}{2B^{h-j+1}}$.

For example, consider a query $QUERY(5, 11)$ on the quantum B+ tree in Figure 1. As mentioned in Section 4.2.1, the candidate nodes are node 1 and node 2. First, we initialize $|\psi\rangle = \frac{1}{\sqrt{2}}(|1\rangle + |2\rangle)$. After applying Hadamard gates and Q_0, we obtain $|\psi\rangle = \frac{1}{\sqrt{8}}(|4\rangle + |5\rangle + |6\rangle + |7\rangle + |8\rangle) + \frac{\sqrt{3}}{\sqrt{8}}|dummy\rangle$, which consists of all the children of node 1 and node 2. Then, after applying Hadamard gates and Q_1, we obtain all the key-record pairs below node 1 and node 2, which is $|\psi\rangle = \frac{1}{\sqrt{32}}(|1\rangle|rec_1\rangle + |2\rangle|rec_2\rangle + |4\rangle|rec_4\rangle + |6\rangle|rec_6\rangle + |8\rangle|rec_8\rangle + |10\rangle|rec_{10}\rangle + |13\rangle|rec_{13}\rangle + |16\rangle|rec_{16}\rangle + |19\rangle|rec_{19}\rangle + |21\rangle|rec_{21}\rangle) + \frac{\sqrt{22}}{\sqrt{32}}|dummy\rangle$. Finally, by a post-selection, we can obtain $|\psi\rangle = \frac{1}{\sqrt{3}}(|6\rangle|rec_6\rangle + |8\rangle|rec_8\rangle + |10\rangle|rec_{10}\rangle)$ with probability $\frac{3}{32}$.

Finally, we show the time complexity of our GCLQ algorithm in the following theorem.

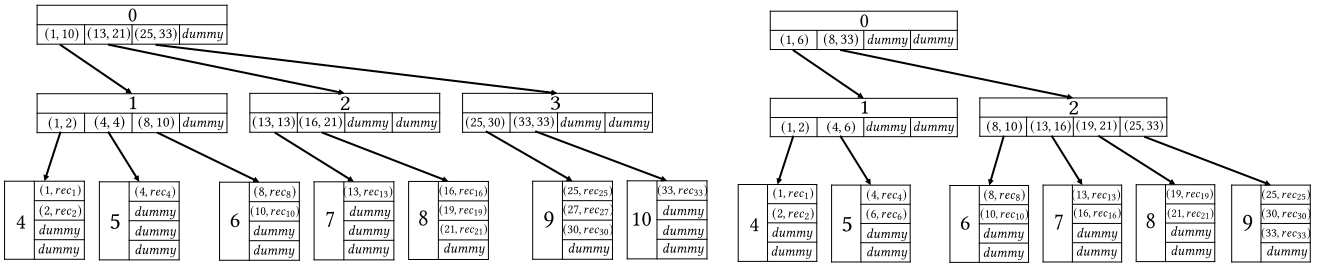
Theorem 4.4. *On average, the quantum range query algorithm returns the answer in $O(C_l \log N)$ time.*

Proof sketch. In the global classical search, we return at most two candidates at each tree level and the height of the tree is $O(\log N)$. Thus, the global classical search involves $O(\log N)$ load operations of QRAM (each has C_l cost) and costs $O(C_l \log N)$ time. In the local quantum search, we need to repeat all the steps in at most $8B$ iterations based on Lemma 4.3, which is a constant number. In each iteration, we repeat Step 2 and Step 3 at most $O(\log N)$ times, and each repetition involves querying a quantum oracle for post-selection. Therefore, the total time complexity is $O(C_l \log N)$. □

Similarly, this time complexity result can be simplified as $O(\log N)$ since C_l can be regarded as a constant and thus skipped according to [31, 34, 36, 39–42]. By Theorem 4.4 and Lemma 4.2, this algorithm is asymptotically optimal in a quantum computer.

5 DYNAMIC VARIANT

In this section, we introduce how to make the static quantum B+ tree dynamic, which solves the dynamic range query. One idea of designing our quantum B+ tree is to retain a classical B+ tree structure and to maintain a concise replication of hierarchical relationships in QRAMs. This enables the flexible extension to the B+ tree variants that have been well studied in classical computers. As such, we adapt the idea of the logarithmic method [11] to enable the insertion operations of the quantum B+ tree by building some forests on the quantum B+ tree. Based on the forests, we propose our approach to perform deletions. In the following, the details



(a) Deletion of $(6, rec_6)$ in node 5: The weight of node 1 becomes 3 which is less than $\frac{1}{4}B^2 = 4$, and thus node 1 becomes imbalanced. It borrows node 6 from node 2 such that node 1 and node 2 are both balanced.

(b) Deletion of $(27, rec_{27})$ in node 9: Node 3 becomes imbalanced, and it cannot borrow node 8 from node 2. Node 2 and node 3 cannot be directly merged since they have 5 children, and thus the whole subtree is rebuilt.

Figure 2: Examples of Deletions

of insertion and deletion are discussed in Section 5.1, and then in Section 5.2, we introduce how to solve the dynamic quantum range query with the dynamic quantum B+ tree.

5.1 Insertion and Deletion

Following the idea of the logarithmic method [11], we build at most L forests, says $F_0, \cdots, F_{L-1}$, where $L = \lfloor \log_B N \rfloor + 1$, and for each $i \in [0, L-1]$, the forest F_i contains at most $B-1$ static quantum B+ trees of height i.

To insert a new key-record pair (key, rec), we insert it into a sorted list first. When the length of this sorted list reaches B, we flush it into F_0. Then, whenever a forest F_i has B quantum B+ trees, which indicates that we have B quantum B+ trees of height i, we merge the B quantum B+ trees of height i into a quantum B+ trees of height $i+1$, and add it into F_{i+1}.

In additional to the classical B+ tree, we also replicate the insertion in the quantum B+ tree such that the hierarchical relationships in the QRAM are consistent with the classical component.

Next, we discuss how to delete a key-record pair in a dynamic quantum B+ tree. It involves two steps. The first step is to locate the quantum B+ tree in the forests which contains the key-record pair. The second step is to delete the key-record pair in both the classical component and the quantum component of the quantum B+ tree.

To locate the tree containing a key-record pair, we assign each key-record pair a unique ID in ascending order (e.g., by a counter). We also maintain an auxiliary B+ tree $\mathcal{T}_0$ to store the one-to-one mapping from the key-record pair to its ID. When a key-record pair is inserted, we insert the key-record pair and its ID into $\mathcal{T}_0$. When a key-record pair is deleted, we delete it in $\mathcal{T}_0$ accordingly. Furthermore, we maintain another auxiliary B+ tree $\mathcal{T}_1$ to store the mapping from the key-record pair ID to the forest F_i it belongs to. Similar to $\mathcal{T}_0$, we do insertions and deletions in $\mathcal{T}_1$ accordingly. In addition, when we merge F_i, we update $\mathcal{T}_1$ for all the key-record pairs in F_i.

To delete a key-record pair (key, rec) in a B+ tree in F_i, we do a classical search to find the leaf that contains (key, rec) and replace the key-record pair with a *dummy*. Then, we check if its ancestors are still balanced. If an imbalanced ancestor is found, we first check if it can borrow a child from its sibling. Figure 2(a) shows an example in this case. Otherwise, we check if it can be directly merged with its sibling, which indicates that the node and its sibling have at most B children. If not, we merge the node and its sibling by rebuilding the subtrees below them. Figure 2(b) shows an example in this case. Then, after rebalancing the B+ tree, we check if the root node still has at least two children. If not, we check if it can borrow a child from another B+ tree in F_i. If not, then if there are at least two B+ trees in F_i, we merge the root nodes of the two B+ trees. Otherwise, we remove the root node and downgrade the B+ tree from F_i to F_{i-1}.

The following lemma shows that the insertions and deletions in the quantum B+ tree is efficient.

Lemma 5.1. *The amortized cost of an insertion and a deletion in a dynamic quantum B+ tree is $O(C_s \log N)$ and $O((C_s + C_l) \log N)$, respectively, where C_s (C_l) is the cost of a store (load) operation on QRAM.*

Proof sketch. By Theorem 3.1 in [11], N insertions totally cost $O(N \log N)$ time. Moreover, each insertion involves store operations on QRAM to at most $L(= \lfloor \log_B N \rfloor + 1)$ forests. Therefore, the amortized cost of one insertion is $O(C_s \log N)$. For deletions, the first step costs $O(C_l \log N)$ time, since it consists of two point queries in B+ trees. By the analysis of partial rebuilding in [45], the second step also costs $O(C_s \log N)$ time (which takes into consideration the store operations on at most L forests similar to insertions). Therefore, the amortized cost of a deletion is $O((C_s + C_l) \log N)$. □

Still, both C_s and C_l can be skipped as they are considered to be constants by [31, 34, 36, 39–42]. Thus, the amortized cost of an insertion and a deletion in a dynamic quantum B+ tree can both be written as $O(\log N)$ for simplicity.

5.2 Query

To answer a query $QUERY(x, y)$ on our dynamic quantum B+ tree, we extend our idea of the GCLQ algorithm to perform both a global classical search and a local quantum search on the forests $F_0, \cdots, F_{L-1}$.

First, for global classical search, we follow the similar global search steps as introduced in Section 4.2 for each tree in the forests. Specifically, we first create an empty list $\mathcal{L}_i$ for each forest F_i ($i \in [0, L-1]$), and we add all the root nodes in F_i into L_i. Then, for each list $\mathcal{L}_i$, we perform a global classical search from each root

in $\mathcal{L}_i$. A candidate set of at most two nodes are returned for each root, which are then added to the original list for easier processing.

Then, we perform the local quantum search for the candidate nodes in the lists $\mathcal{L}_0, \cdots, \mathcal{L}_{L-1}$. Consider the nodes $u_0, u_1, \cdots, u_{m-1}$ in the lists, where m is the total number of nodes in $\mathcal{L}_0, \cdots, \mathcal{L}_i$. We initialize the quantum bits to be $\sum_{i=0}^{m-1} \frac{\sqrt{B^{h(u_i)+1}}}{\sqrt{\sum_{j=0}^{m-1} B^{h(u_j)+1}}} |u_i\rangle$, where $h(u_i)$ is the height of u_i, and $\frac{\sqrt{B^{h(u_i)+1}}}{\sqrt{\sum_{j=0}^{m-1} B^{h(u_j)+1}}}$ is the normalized amplitude of each u_i such that each of the key-record pair below the nodes in the lists has the same amplitude in the result. Then, we do the same steps in Section 4.2.2 to obtain the leaves below the nodes. Finally, we do a post-selection search as discussed in Section 4.2.

The following theorem shows the time complexity for the dynamic quantum B+ tree to answer a quantum range query.

Theorem 5.2. *On average, the dynamic quantum B+ tree answers a range query in $O(C_l \log^2 N)$ time.*

Proof sketch. The global classical search costs $O(C_l \log^2 N)$ time, since we have $O(\log N)$ forests and the global search for each forest needs $O(C_l \log N)$ time. For the local quantum search, the initialization and the steps to obtain the leaves cost $O(C_l \log N)$. The average number of post-selection is $O(\log N)$, since there are at most $2B\lfloor \log_B N \rfloor$ nodes in the lists. Totally, the time complexity is $O(C_l \log^2 N)$. □

Similarly, the time complexity $O(C_l \log^2 N)$ can be simplified as $O(\log^2 N)$ by [31, 34, 36, 39–42].

6 EXPERIMENT

In this section, we show our experimental results to verify the superiority of our proposed quantum B+ tree on quantum range queries. The study of the real-world quantum supremacy [7, 13, 55], which is to confirm that a quantum computer can do tasks faster than classical computers, is still a developing topic in the quantum area. Thus, in this paper, we would not verify the quantum supremacy, which will be fully verified in a future quantum computer.

In this paper, we have two measurements. (a) *IO Cost:* All quantum-based and classical-based algorithms could be measured by the *IO cost*. Following a well-established common practice in the database community of measuring the performance of an operator (e.g., the cost of building an index [44, 61, 64], the cost of performing a query over an index [10, 33, 64] and the estimated cost of an evaluation plan for query optimization [17]) using the number of page accesses (simply the *IO cost*), we chose to evaluate the *number of memory accesses* as the *IO cost* to make the comparisons. In the quantum data structures, a QRAM read or write operation is counted as 1 IO, because researchers believe that with the future advances of quantum computers, the store and load operations of QRAM can be very efficient in the form of quantum oracles [31, 34, 36, 39–42]. In the classical data structures, a page access is counted as 1 IO as well.

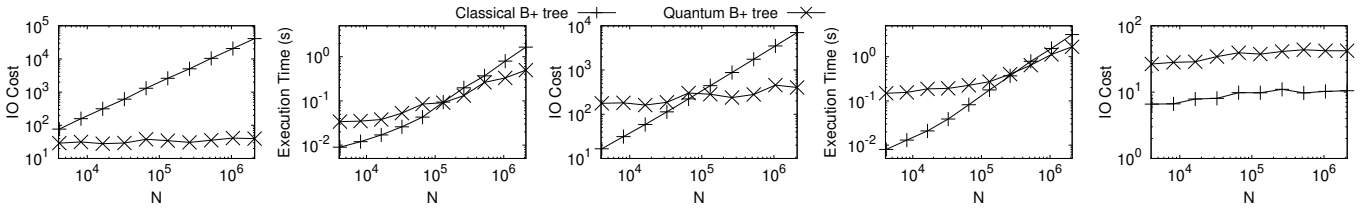
(b) *Execution Time:* Though the IO cost is an important measurement in the database community, it is also interesting to measure the execution time of all quantum-based and classical-based algorithms even if the *practical* implementation quantum computer involving a lot of Qubits is still under development. Specifically, although there exist some quantum simulators such as Qiskit [59] and Cirq [25], they are not capable of simulating an efficient QRAM for executing our quantum range queries, because these simulators can hold a very limited number of qubits, which support only queries with only several data records, each with at most 2-3 bits. Thus, we chose to use C++ to perform the quantum simulations. We estimate the execution time of each of our quantum-based algorithms as follows. Consider a quantum-based algorithm. We perform the following steps.

(1) We construct all the quantum oracles (in the form of quantum circuits) involved in the execution of quantum range queries.
(2) In each of the above quantum oracles, we obtain a list of all quantum gates of various types that need to be run sequentially.
(3) We measure the running time of each type of a quantum gate in a real circuit simulated by Qiskit [59].
(4) We estimate the running time of each quantum oracle by summing up the measured running time of each gate.

Finally, the execution time of our quantum range queries is estimated to be the combination of the original running time (of our C++ quantum simulation) and the running time of all quantum oracles. Note that for each QRAM store or load operation, we also follow the above steps to construct its quantum oracle, where we use a state-of-the-art efficient quantum circuit implementation of the Bucket Brigade QRAM model [47]. In the classical data structures, we measure the real execution time for each query. The PC we used is equipped with an Apple M3 Pro CPU and 18GB memory.

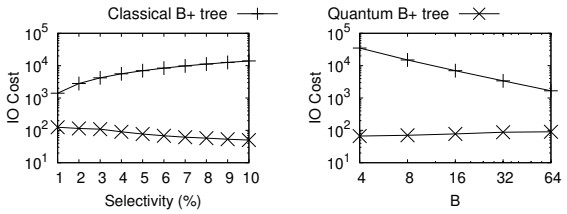
We used two real datasets from SNAP [37], namely *Brightkite* and *Gowalla*. Each of the two datasets contains a set of check-in records, each of which consists of a timestamp (i.e., an integer) and a location (i.e., a 2-tuple of integers). The original sizes of the two datasets are 4M and 6M, respectively. We set the timestamp as the search key. We also used the *IMDb* dataset [1] containing a set of 1.4M movie records, each of which consists of a release year (i.e., an integer) and a rating (i.e., a real number). For this dataset, we answer a query of finding the movie with release year between a range that has the highest rating, which is aimed at verifying the superiority of using our quantum range query result (i.e., the movie candidates with a specific release year range) in superposition as the input to the quantum algorithm that finds the item with the highest rating in the database.

We compare our GCLQ search algorithm on our proposed quantum B+ tree and its dynamic variant (which are simply denoted as **quantum B+ tree**) with the classical B+ tree search algorithm and its dynamic version [11] (which are denoted as **classical B+ tree**). For the queries on the IMDb dataset, we compare the following approaches. (1) **Quantum maximum search with quantum B+ tree**: it uses our quantum B+ tree to find the result of movie candidates in superposition and then uses a quantum maximum search [16] to find the movie with the highest rating. (2) **Quantum maximum search with classical B+ tree**: it uses the classical B+ tree to find the result of movie candidates in classical bits and then uses the quantum maximum search to find the movie with the highest rating. (2) **Linear scan with classical B+ tree**: it uses the classical B+ tree to find the result of movie candidates in classical bits and then performs linear scan to find the movie with the highest rating.



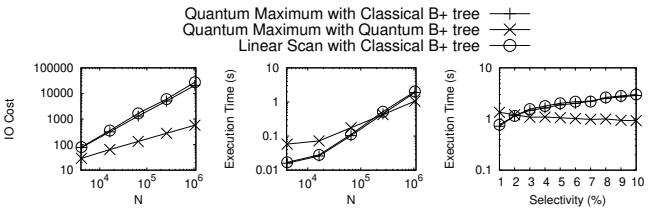
(a) *Brightkite* (b) *Brightkite* (c) *Brightkite* (d) *Brightkite* (e) *Brightkite*

Figure 3: The Effect of N on (a) & (b) Quantum Range Queries, (c) & (d) Dynamic Range Queries and (e) Insertions and Deletions into the Dynamic Data Structures on Dataset *Brightkite*



(a) *Brightkite* (b) *Brightkite*

Figure 4: The Effect of (a) Selectivity and (b) B on Quantum Range Queries on Dataset *Brightkite*



(a) *IMDb* (b) *IMDb* (c) *IMDb*

Figure 5: The Effect of N and Selectivity on Database Maximum Query on Dataset *IMDb*

The factors we studied are the dataset size N, the branching factor B and the selectivity (which is defined to be the proportion of items within the query range among the entire dataset). For datasets *Brightkite* and *Gowalla*, we varied N from 4K to 2M (by randomly choosing a subset from each dataset with the target size). We varied B from 4 to 64 for each data structure. We varied the selectivity from 1% to 10%. By default, N = 2M, B = 16 and the selectivity is 5%. For datasets *IMDb* which has smaller size, the only difference is that we set the maximum N and default N to be 1M. For each type of range query, we randomly generate 10,000 range queries and report the average measurement.

We present our experimental results as follows. For the sake of space, we mainly show the results of dataset *Brightkite* and *IMDb*, while we observe similar results in the other dataset *Gowalla* in all our experiments. The complete results of dataset *Gowalla* can be found in our technical report [8].

Effect of N. We first study the effect of the dataset N, which verifies the scalability of our proposed data structures and algorithms. As shown in Figures 3(a) and (b), our proposed **quantum B+ tree** with our proposed GCLQ search algorithm scales well when N grows from 4K to 2M on both datasets, which verifies the efficient $O(C_l \log_B N)$ cost of performing the GCLQ search on our **quantum B+ tree**. As N increases, the number of items within the query range (i.e., k) for each query also tend to increase (approximately linearly with N). Thus, the IO cost for the **classical B+ tree** demonstrates a linear growth with N, which complies its $O(\log N + k)$ complexity. In comparison, the performance of our **quantum B+ tree** does not depend on the number of range query results, and thus it has up to 1000x more efficient IO cost than the **classical B+ tree** for the (static) quantum range queries.

In particular, the estimated execution time of using our **quantum B+ tree** also demonstrates only slight increase when N increases. This is because the cost of load operations on QRAM is not significant affected by the database size as well. When N increases to 2M, the estimated execution time of **quantum B+ tree** is 3.3x faster than the classical B+ tree.

For the dynamic quantum range queries, although our (dynamic) **quantum B+ tree** has higher IO cost and execution time than the (dynamic) **classical B+ tree** for small N (i.e., < 100K), it shows much better scalability, as illustrated in Figure 3(c) and (d).

For the dynamic versions, we also tested the performance of insertions and deletions. Following [4], we insert all the records in the datasets where for each insertion, there is 1% chance to delete an existing record instead of doing this insertion, and we measure IO cost for each insertion or deletion operation on average. Figure 3(e) shows that the dynamic **quantum B+ tree** needs 3x higher IO cost for insertion or deletion, because the **quantum B+ tree** has more complex structure than the **classical B+ tree**. However, the **quantum B+ tree** shows the $O(N)$ growth, which is similar to the **classical B+ tree**, indicating that the insertion and deletions operations on our **quantum B+ tree** are still reasonably efficient.

Effect of Selectivity. Then, we study the effect of selectivity on the quantum range queries. When the selectivity (i.e., $\frac{k}{N}$) increases, as shown in Figures 4(a), the IO cost increases linearly for the **classical B+ tree** due to the $O(\log N + k)$ query complexity. Our proposed **quantum B+ tree** with $O(\log N)$ cost surprisingly shows better performance as the selectivity increases. This is because a larger k expedites the classical global search in our GCLQ search algorithm, such that the efficient local quantum search can be triggered earlier. When k increases, the post-selection will also be accelerated since the cost of post-selection is linear to $\frac{N}{k}$ as we mentioned in Section 4.2. We observe similar results on the dynamic case.

Effect of B. We also study the effect of the branching factor B. As shown in Figure 4(b), when B increases, the IO cost for the classical data structures decreases sharply, since the height of the tree is smaller for larger B. Our quantum data structures needs slightly higher IO cost as B increases, because the success rate of the post-selection could be affected as we mentioned in Section 4.2.2. However, it is well known that a larger branching factor

leads to more memory consumption for a classical B+ tree [18] (and thus a quantum B+ tree as well). Thus, we set B to be 16 in other experiments for fair comparisons with a reasonable memory consumption. Overall, our proposed quantum data structures favor a smaller branching factor. Similar results are also observed on the dynamic case.

Maximum Query. We show the result of maximum queries on dataset *IMDb*. As shown in Figure 5(a) and (b), using the quantum maximum search and our **quantum B+ tree** achieves the best scalability in IO cost and execution time when N increases. This is because our **quantum B+ tree** returns quantum range queries results (i.e., the movie candidates in the query range) very efficiently and the results are in the form of superposition (without the need to list out all of them) which can be directly input to the quantum maximum search. Moreover, the quantum maximum search is also faster than the linear scan since it only costs $O(\sqrt{N})$ based on [16]. Note that with quantum maximum search and the classical B+ tree, the queries are still less efficient since the cost of returning all the range queries result is obvious.

Summary. In summary, our **quantum B+ tree** with our proposed GCLQ search algorithm achieves up to 1000x performance improvement than the **classical B+ tree**. On the dataset *Brightkite* of size 2M, the average IO cost is only around 40 for the **quantum B+ tree** for the quantum range query, while the **classical B+ tree** needs around 40K IO cost. The similar superiority of our **quantum B+ tree** is observed on the dynamic quantum range queries compared with the classical data structures. We also show that our **quantum B+ tree** scales well with the dataset size, the selectivity of the query ranges and the branching factor.

7 RELATED WORK

The classical range query problem has been studied for decades with various data structures proposed to solve this problem. There is little room for further significant improvement. For the static range query and the dynamic range query, the B-tree [9] and the B+ tree [18] are widely-used. The B+ tree is a tree data structure which can find a key in $O(\log N)$ time, where N is the number of key-record pairs and B is the branching factor. Then, $O(k)$ time is needed to load the k key-record pairs in the query range, so the range query on a B+ tree costs $O(\log N + k)$ time, which is shown to be asymptotically optimal in classical computers [62].

However, all the above classical data structures have the same problem that the execution time grows linearly with k, which makes them useless for quantum algorithms.

The quantum database searching problem is also very popular in the quantum algorithm field. Grover's algorithm is described as a database search algorithm in [26]. It solves the problem of searching a marked record in an unstructured list, which means that all the N records are arranged in random order. On average, the classical algorithm needs to perform $N/2$ queries to a function f which tells us if the record is marked. More formally, for each index i, $f(i) = 1$ means the i-th record is marked and $f(i) = 0$ means the i-th record is unmarked. Note that only one record is marked in the database. Taking the advantage of quantum parallelism, Grover's algorithm can find the index of the marked record with $O(\sqrt{N})$ queries to the oracle. The main idea is to first "flip" the amplitude of the answer state and then reduce the amplitudes of the other states. One such iteration will enlarge the amplitude of the answer state and $O(\sqrt{N})$ iterations should be performed until the probability that the qubits are measured to be the answer comes close to 1. Then, an improved Grover's algorithm was proposed in [14]. We are also given the function f to mark the record, but k records are marked at this time. The improved Grover's algorithm can find one of the k marked records in $O(\sqrt{N/k})$ time. If we make the function f to mark the records with keys within a query range, then this algorithm returns one of the k key-record pairs in $O(\sqrt{N/k})$ time, and it needs $O(\sqrt{Nk})$ time to answer a range query. Since the query time grows linearly with the square root of N, this algorithm is much less efficient when N is very large, even compared to using the classical data structure with $O(\log N + k)$ time complexity. The reason is that this algorithm only handle the unstructured dataset and does not leverage the power of data structures that could be pre-built as a database index.

In the database area, there are also plenty of studies discussing how to use quantum computers to further improve traditional database queries. For example, [23, 51, 56, 57] discussed quantum query optimization, which is to use quantum algorithms like quantum annealing [24] to optimize a traditional database query. However, compared with the quantum range query discussed in this paper, the existing studies are in a different direction. The existing studies are discussing how to use a quantum algorithm to optimize range queries in a classical computer, where the query returns a list of records. In this paper, we discuss how to use a classical algorithm to optimize range queries in a quantum computer, where the query returns quantum bits in a superposition of records. We focus on a quantum data structure stored in a quantum computer, where the classical algorithm is only to assist the query. To our best knowledge, in the database area, we are the first to discuss quantum algorithms in this direction.

In conclusion, the existing classical data structures and the existing quantum database searching algorithms cannot solve the range query problem in quantum computers perfectly.

8 CONCLUSION

In this paper, we study the quantum range query problem. We propose the quantum B+ tree, the first tree-like quantum data structure, and the efficient global-classical local-quantum search algorithm based on the quantum B+ tree. Our proposed data structure and algorithm can answer a quantum range query in $O(\log N)$ time, which is asymptotically optimal in quantum computers and is exponentially faster than classical B+ trees. Furthermore, we extend it to a dynamic quantum B+ tree. The dynamic quantum B+ tree can support insertions and deletions in $O(\log N)$ time and answer a dynamic quantum range query in $O(\log^2 N)$ time. In the experiments, we did simulations to verify the superiority of our proposed quantum data structures compared with the classical data structures. We expect that the quantum data structures will show significant advantages in the real world.

The future work includes exploring even more efficient quantum algorithms for the dynamic and studying more advanced database queries such as the top-k queries.

REFERENCES

[1] 2025. IMDb Non-Commercial Datasets. https://developer.imdb.com/non-commercial-datasets/.

[2] Soumik Adhikary, Siddharth Dangwal, and Debanjan Bhowmik. 2020. Supervised learning with a quantum classifier using multi-level systems. *Quantum Information Processing* 19, 3 (2020), 1–12.

[3] Ashish Ahuja and Sanjiv Kapoor. 1999. A quantum algorithm for finding the maximum. *arXiv preprint quant-ph/9911082* (1999).

[4] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. 2014. Storage management in AsterixDB. *Proceedings of the VLDB Endowment* 7, 10 (2014), 841–852.

[5] Andris Ambainis. 1999. A better lower bound for quantum algorithms searching an ordered list. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 352–357.

[6] Lars Arge and Jeffrey Scott Vitter. 1996. Optimal dynamic interval management in external memory. In *Proceedings of 37th Conference on Foundations of Computer Science*. IEEE, 560–569.

[7] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[8] Anonymous Author(s). 2023. *First Tree-like Quantum Data Structure: Quantum B+ Tree*. Technical Report. https://github.com/anonym45263/EBDE402C7C7404A8

[9] Rudolf Bayer and Edward McCreight. 2002. Organization and maintenance of large ordered indexes. In *Software pioneers*. Springer, 245–262.

[10] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[11] Jon Louis Bentley and James B Saxe. 1980. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms* 1, 4 (1980), 301–358.

[12] Axel Berg, Magnus Oskarsson, and Mark O'Connor. 2021. Deep ordinal regression with label diversity. In *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2740–2747.

[13] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. 2018. Characterizing quantum supremacy in near-term devices. *Nature Physics* 14, 6 (2018), 595–600.

[14] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. 1998. Tight bounds on quantum searching. *Fortschritte der Physik: Progress of Physics* 46, 4-5 (1998), 493–505.

[15] Gilles Brassard, Peter Høyer, and Alain Tapp. 1998. Quantum counting. In *International Colloquium on Automata, Languages, and Programming*. Springer, 820–831.

[16] Yanhu Chen, Shijie Wei, Xiong Gao, Cen Wang, Jian Wu, and Hongxiang Guo. 2019. An optimized quantum maximum or minimum searching algorithm and its circuits. *arXiv preprint arXiv:1908.07943* (2019).

[17] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. 2001. Query optimization in compressed database systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 271–282.

[18] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[19] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[20] Paul Adrien Maurice Dirac. 1939. A new notation for quantum mechanics. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 35. Cambridge University Press, 416–418.

[21] James Dougherty, Ron Kohavi, and Mehran Sahami. 1995. Supervised and unsupervised discretization of continuous features. In *Machine learning proceedings 1995*. Elsevier, 194–202.

[22] Christoph Durr and Peter Hoyer. 1996. A quantum algorithm for finding the minimum. *arXiv preprint quant-ph/9607014* (1996).

[23] Tobias Fankhauser, Marc E Solèr, Rudolf M Füchslin, and Kurt Stockinger. 2021. Multiple query optimization using a hybrid approach of classical and quantum computing. *arXiv preprint arXiv:2107.10508* (2021).

[24] Aleta Berk Finnila, MA Gomez, C Sebenik, Catherine Stenson, and Jimmie D Doll. 1994. Quantum annealing: A new method for minimizing multidimensional functions. *Chemical physics letters* 219, 5-6 (1994), 343–348.

[25] Bacon D Gidney C and contributors. 2018. Cirq: A python framework for creating, editing, and invoking noisy intermediate scale quantum (NISQ) circuits. https://github.com/quantumlib/Cirq.

[26] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 212–219.

[27] Lov K Grover and Jaikumar Radhakrishnan. 2005. Is partial quantum search of a database any easier?. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. 186–194.

[28] Anant Gupta and Kuldeep Singh. 2013. Location based personalized restaurant recommendation system for mobile environments. In *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 507–511.

[29] Jacques Hadamard. 1893. Resolution d'une question relative aux determinants. *Bull. des sciences math.* 2 (1893), 240–246.

[30] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum algorithm for linear systems of equations. *Physical review letters* 103, 15 (2009), 150502.

[31] Akinori Hosoyamada and Yu Sasaki. 2018. Quantum Demiric-Selçuk meet-in-the-middle attacks: applications to 6-round generic Feistel constructions. In *International Conference on Security and Cryptography for Networks*. Springer, 386–403.

[32] Nabil Ibtehaz, M Kaykobad, and M Sohel Rahman. 2021. Multidimensional segment trees can do range updates in poly-logarithmic time. *Theoretical Computer Science* 854 (2021), 30–43.

[33] Christian S Jensen, Dan Lin, and Beng Chin Ooi. 2004. Query and update efficient B+-tree based indexing of moving objects. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 768–779.

[34] Ruslan Kapralov, Kamil Khadiev, Joshua Mokut, Yixin Shen, and Maxim Yagafarov. 2020. Fast Classical and Quantum Algorithms for Online k-server Problem on Trees. *arXiv preprint arXiv:2008.00270* (2020).

[35] Iordanis Kerenidis and Anupam Prakash. 2017. Quantum Recommendation Systems. In *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[36] Maria Kieferova, Ortiz Marrero Carlos, and Nathan Wiebe. 2021. Quantum Generative Training Using R\'enyi Divergences. *arXiv preprint arXiv:2106.09567* (2021).

[37] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.

[38] Guangxi Li, Zhixin Song, and Xin Wang. 2021. VSQL: variational shadow quantum learning for classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 8357–8365.

[39] Tongyang Li, Shouvanik Chakrabarti, and Xiaodi Wu. 2019. Sublinear quantum algorithms for training linear and kernel-based classifiers. In *International Conference on Machine Learning*. PMLR, 3815–3824.

[40] Tongyang Li, Chunhao Wang, Shouvanik Chakrabarti, and Xiaodi Wu. 2021. Sublinear Classical and Quantum Algorithms for General Matrix Games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 8465–8473.

[41] Ashley Montanaro. 2017. Quantum pattern matching fast on average. *Algorithmica* 77, 1 (2017), 16–39.

[42] María Naya-Plasencia and André Schrottenloher. 2020. Optimal Merging in Quantum k-xor and k-sum Algorithms. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 311–340.

[43] Michael A Nielsen and Isaac L Chuang. 2001. Quantum computation and quantum information. *Phys. Today* 54, 2 (2001), 60.

[44] Patrick O'Neil and Dallan Quass. 1997. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 38–49.

[45] Mark H Overmars. 1983. *The design of dynamic data structures*. Vol. 156. Springer Science & Business Media.

[46] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[47] Alexandru Paler, Oumarou Oumarou, and Robert Basmadjian. 2020. Parallelizing the queries in a bucket-brigade quantum random access memory. *Physical Review A* 102, 3 (Sept. 2020). https://doi.org/10.1103/physreva.102.032608

[48] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. 2014. Quantum support vector machine for big data classification. *Physical review letters* 113, 13 (2014), 130503.

[49] Yue Ruan, Xiling Xue, Heng Liu, Jianing Tan, and Xi Li. 2017. Quantum algorithm for k-nearest neighbors classification based on the metric of hamming distance. *International Journal of Theoretical Physics* 56, 11 (2017), 3496–3507.

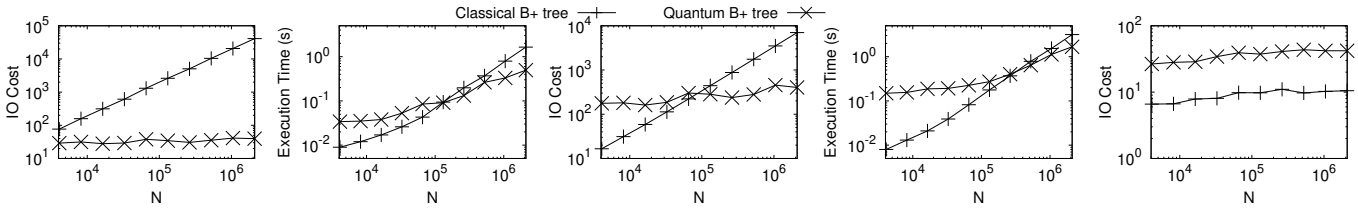
[50] Seyran Saeedi and Tom Arodz. 2019. Quantum sparse support vector machines. *arXiv preprint arXiv:1902.01879* (2019).

[51] Manuel Schönberger. 2022. Applicability of quantum computing on database query optimization. In *Proceedings of the 2022 International Conference on Management of Data*. 2512–2514.

[52] Erwin Schrödinger. 1935. Discussion of probability relations between separated systems. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 31. Cambridge University Press, 555–563.

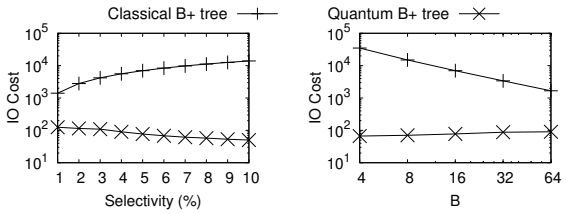
[53] Pranab Sen and Srinivasan Venkatesh. 2001. Lower bounds in the quantum cell probe model. In *International Colloquium on Automata, Languages, and Programming*. Springer, 358–369.

[54] Peter W Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 124–134.



(a) *Gowalla* (b) *Gowalla* (c) *Gowalla* (d) *Gowalla* (e) *Gowalla*

Figure 6: The Effect of N on (a) & (b) Quantum Range Queries, (c) & (d) Dynamic Range Queries and (e) Insertions and Deletions into the Dynamic Data Structures on Dataset *Gowalla*



(a) *Gowalla* (b) *Gowalla*

Figure 7: The Effect of (a) Selectivity and (b) B on Quantum Range Queries on Dataset *Gowalla*

[55] Barbara M Terhal. 2018. Quantum supremacy, here we come. *Nature Physics* 14, 6 (2018), 530–531.

[56] Immanuel Trummer and Christoph Koch. 2015. Multiple query optimization on the D-Wave 2X adiabatic quantum computer. *arXiv preprint arXiv:1510.06437* (2015).

[57] Valter Uotila. 2022. Synergy between Quantum Computers and Databases. (2022).

[58] Nathan Wiebe, Ashish Kapoor, and Krysta M Svore. 2015. Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning. *Quantum Information & Computation* 15, 3-4 (2015), 316–356.

[59] Robert Wille, Rod Van Meter, and Yehuda Naveh. 2019. IBM's Qiskit tool chain: Working with and developing for real quantum computers. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1234–1240.

[60] Yanbing Xue and Milos Hauskrecht. 2017. Efficient learning of classification models from soft-label information by binning and ranking. In *The Thirtieth International Flairs Conference*.

[61] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the design of LSM-tree based OLTP storage engine with persistent memory. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1872–1885.

[62] Andrew Chi-Chih Yao. 1981. Should tables be sorted? *Journal of the ACM (JACM)* 28, 3 (1981), 615–628.

[63] Kun Zhang and Vladimir Korepin. 2018. Quantum partial search for uneven distribution of multiple target items. *Quantum Information Processing* 17, 6 (2018), 1–20.

[64] Rui Zhang, Jianzhong Qi, Dan Lin, Wei Wang, and Raymond Chi-Wing Wong. 2012. A highly optimized algorithm for continuous intersection join queries over moving objects. *The VLDB journal* 21 (2012), 561–586.

A REMAINING EXPERIMENTAL RESULTS

In this section, we show the complete experimental results of dataset *Gowalla*, which are not presented in Section 6 due to the space limit.

As shown in Figures 6 and 7, we observe similar trends and results for dataset *Gowalla* as dataset *Brightkite*, which further verifies the efficiency of our proposed quantum B+ tree on quantum range queries.

B COMPLETE PROOFS

In this section, we show the complete proofs of all the lemmas and theorems in this paper.

Proof of Lemma 4.2. Consider a static quantum range query $QUERY(x, x)$. It answers whether the key x exists in the dataset, which is a static membership problem. By [53], the time complexity of answering a static membership problem in quantum computers is $\Omega(\log N)$. Therefore, the time complexity of answering a static quantum range query is also $\Omega(\log N)$. □

Proof of Lemma 4.3. Since the routing keys of all nodes in the same level are disjoint, we cannot have more than two partial nodes in the same level. It is also easy to verify that the returned nodes are from the same level, Thus, the candidate set contains at most two nodes. Since we only filter out the outside node in this algorithm, we have $\mathcal{R}^* \subset \mathcal{R}$. Finally, since the precise partial node either is a leaf (which contains at least $1/B$ items in the query range), or contains an inside child (which contains at least $\frac{1}{4B}$ items in the query range due to the balanced nodes of the B+ tree), and it can be verify that at least one returned node is precise, we have $|\mathcal{R}^*| \geq \frac{1}{8B}|\mathcal{R}|$. □

Proof of Lemma 4.4. The global classical search costs $O(\log N)$ time because we return at most two candidates and the height of the tree is $O(\log N)$. For the local quantum search, we repeat all the steps for $\frac{2B^{h-j+1}}{k}$ times on average. By the condition to trigger a local quantum search, all the non-dummy key-record pairs below one of the children of u and v are all in the answer, therefore $k \geq \frac{1}{4}B^{h-j}$ by the definition of our quantum B+ tree. So, we need to repeat all the steps for at most $8B$ times, which is a constant time. In each iteration, we do Step 2 and Step 3 for at most $O(\log_B N)$ times, so the local quantum search needs $O(\log_B N)$ time. Therefore, the quantum range query algorithm needs $O(\log N)$ time. □

Proof of Theorem 5.1. By Theorem **??** in this paper and Theorem 3.1 in [11], N insertions totally cost $O(N \log_B N)$ time. Therefore, the amortized cost of one insertion is $O(\log_B N)$.

To analyze the update cost in a merge, we find that when merging F_i, the time complexity to update T_1 is $O(\log_B N)$. The reason is as follows.

Let ID_l denote the smallest ID in F_i. For each $j < i$ and each key-record pair in F_j, the ID of the key-record pair is smaller than ID_l. Let ID_r denote the largest ID in F_i. Then, for each $j > i$ and each key-record pair in F_j, the ID of the key-record pair is greater than ID_r. Therefore, the update operations for the key-record pairs in F_i can be merged into a range update. Then, we can use lazy propagation [32] to do the range update in $O(\log N)$ time.

Thus, the extra cost to maintain T_0 and T_1 has no impact, which means the amortized cost of insertion is still $O(\log N)$.

For deletions, the first step costs $O(\log N)$ time, since it consists of two point queries in B+ trees.

Then consider the second step. Motivated by the analysis of partial rebuilding in [45], we consider a node u of height h just after a rebuild. The node u is perfectly balanced such that its weight $w(u) \geq \frac{1}{2}B^{h+1}$. Since it will become imbalanced if and only if $w(u) < \frac{1}{4}B^{h+1}$, there must be $\Omega(B^{h+1})$ deletions below the node u or its siblings before that. So, it is charged $O(1)$ time for each deletion below it and its siblings. Then, for a deletion in a leaf node, each ancestor and its siblings are charged $O(1)$ time, so they are totally charged $O(\log N)$ time.

Therefore, the amortized cost of a deletion is $O(\log N)$.

□

Proof of Theorem 5.2. As previously mentioned, the global classical search costs $O(\log^2 N)$ time. Consider the cost of the local quantum search. The initialization and the steps to obtain the leaves cost $O(\log N)$. Then, we multiply it by the average post-selection times. Let N' denote the total number of key-record pairs below the nodes in $L_0, \cdots, L_i$. Then, the post-selection needs to be iterated $O(N'/k)$ times on average. Since there are at most $2B\lfloor \log_B N \rfloor$ nodes in the lists and each node has a weight at most B^{i+1}, we have $N' \leq 2B^{i+2} \log_B N$. Before we turn to the local quantum search, we have found a child of a node in L_i such that all the key-record pairs below the child are in the answer, such that $k \geq \frac{1}{4}B^i$. Therefore, $N'/k = O(\log_B N)$. Hence, the local quantum search costs $O(\log^2 N)$ time on average.

Since both the global classical search and the local quantum search cost $O(\log^2 N)$ time, the dynamic quantum B+ tree answers a range query in $O(\log^2 N)$ time on average. □