Waffle: In-memory Grid Index for Moving Objects with Reinforcement Learning-based Configuration Tuning System

Dalsu Choi
Korea University
Seoul, Korea
dalsuchoi@korea.ac.kr

Hyunsik Yoon
Korea University
Seoul, Korea
hyunsikyoon@korea.ac.kr

Hyubjin Lee
Korea University
Seoul, Korea
hyubjinlee@korea.ac.kr

Yon Dohn Chung*
Korea University
Seoul, Korea
ydchung@korea.ac.kr

ABSTRACT

Location-based services for moving objects are close to our lives. For example, ride-sharing services, micro-mobility services, navigation and traffic management, delivery services, and autonomous driving are all based on moving objects. The efficient management of such moving objects is therefore getting more and more important. The main challenge is the handling of a large number of location-update queries with scan queries. To address this challenge, we propose a novel in-memory grid indexing system, Waffle, for moving objects. Waffle divides a geographical space into fixed-sized cells. For efficient query processing, Waffle forms chunks, each of which consists of neighboring cells. Such a Waffle index is defined by several configuration knobs. A knob configuration has a significant impact on the performance of Waffle, and an appropriate configuration may change as objects continuously move. Therefore, we propose an online configuration tuning system, WaffleMaker, that automatically determines not only knob values but also when to change knob values, as a part of Waffle. Using a configuration determined by WaffleMaker, Waffle rebuilds the current index without blocking user queries based on a concurrency control scheme. Through extensive experiments, we show that Waffle performed better than the existing methods, and WaffleMaker automatically tuned configuration knob values.

PVLDB Reference Format:
Dalsu Choi, Hyunsik Yoon, Hyubjin Lee, and Yon Dohn Chung. Waffle: In-memory Grid Index for Moving Objects with Reinforcement Learning-based Configuration Tuning System. PVLDB, 15(11): 2375-2388, 2022.
doi:10.14778/3551793.3551800

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dalsuchoi/waffle.

1 INTRODUCTION

A lot of location-based services based on moving objects are getting more and more popular and important. For example, ride-sharing services (e.g., Uber and Lyft), micro-mobility services (e.g., Lime and Bird), navigation and traffic management systems (e.g., Google Maps and Waze), delivery services (e.g., Grubhub, DoorDash, Uber eats, and Postmates), and autonomous driving (e.g., Tesla) are all based on moving objects. Therefore, the efficient management of these moving objects is one of the important challenges in real-world services.

The main characteristic of moving objects is that their positions defined by latitude and longitude continuously change. Therefore, the support of rapid location-updates is the primary requirement. Furthermore, the fast processing of scan queries including range and k-NN queries is another requirement for retrieving necessary information for the services.

In this paper, we propose a novel in-memory grid indexing system, **Waffle**, for moving objects. The grid coordinate of an object is easily calculated based on a grid definition, which is similar to finding a target bucket in a hash data structure. The feature constructs the basis of efficient location-update query processing. Moreover, an in-memory index provides a fast response time, which is critical for real-world services.

A Waffle index is defined by several configuration knobs, which have a significant impact on the performance of Waffle. Appropriate knob values depend on various factors including object distribution, query workload, hardware configuration, and a trade-off between query processing time and memory usage.

Toward appropriate configuration, we propose an online configuration tuning system, **WaffleMaker**, as a component of Waffle. WaffleMaker is based on reinforcement learning, does not require pre-training, and models the following process. WaffleMaker observes the current object distribution, outputs a knob setting, and obtains the performance results for the Waffle index defined by the knob setting. By repeating the process, WaffleMaker gradually outputs a configuration that leads to better performance.

Whenever given a new configuration automatically determined by WaffleMaker, Waffle redefines a Waffle index, called a **regrid**. While rebuilding a Waffle index, without a mechanism to handle user queries, the query processing is blocked, and the blocked time increases when the number of objects is large. Waffle overcomes the problem by including a concurrency control scheme.

Our contributions are summarized as follows. First, we propose a novel in-memory grid indexing system, Waffle, for moving objects. We propose the details on how to organize the objects in the main memory based on the concept of cells and chunks. Second, we propose a novel reinforcement learning-based configuration tuning system, WaffleMaker. Third, we propose a novel workflow model to redefine a grid index without blocking user queries. Finally, we perform extensive experiments to demonstrate the efficiency of Waffle.

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551800

The rest of this paper is organized as follows. Section 2 introduces the basic structure of a Waffle grid index, and Section 3 describes the query processing. We then discuss effects of knob configuration in Section 4 and a regrid mechanism in Section 5. Next, Section 6 introduces WaffleMaker, an online configuration tuning system. In Section 7, we describe the experimental evaluation. Section 8 introduces previous related studies, and Section 9 concludes the paper.

2 BASIC INDEX STRUCTURE

We introduce the basic structure of a Waffle grid index based on a concept of cells and chunks. A **geographical space** is defined as $([min_{lat}, max_{lat}), [min_{lon}, max_{lon}))$, where min_{lat} and max_{lat} are the minimum and maximum latitude values, and min_{lon} and max_{lon} are the minimum and maximum longitude values, respectively. The **coordinate of an object** O in the space is represented by (O_{lat}, O_{lon}).

*Definition 2.1 (**Cell**).* Suppose that (1) the number of cells along latitude in the space is $\boldsymbol{nCell_{lat}^{space}}$, and (2) the number of cells along longitude in the space is $\boldsymbol{nCell_{lon}^{space}}$. The coordinate of an object O is mapped to the integer **cell coordinate**, $(cell_{lat}, cell_{lon})$, where

$$cell_{lat} = \lfloor \frac{O_{lat} - min_{lat}}{max_{lat} - min_{lat}} \times nCell_{lat}^{space} \rfloor$$
$$cell_{lon} = \lfloor \frac{O_{lon} - min_{lon}}{max_{lon} - min_{lon}} \times nCell_{lon}^{space} \rfloor. \quad (1)$$

A **cell** of Waffle, $cell(cell_{lat}, cell_{lon})$, is a set of objects with the same cell coordinate, and the number of objects in a cell is limited by $\boldsymbol{MOPC}$ (Maximum Objects Per Cell). □

A cell with zero object is called an empty cell. $nCell_{lat}^{space}$, $nCell_{lon}^{space}$, and $MOPC$ are the knobs of Waffle used to construct cells. If more than $MOPC$ objects have the same cell coordinate, Waffle creates more than one cell, and we describe the detailed query processing in Section 3.

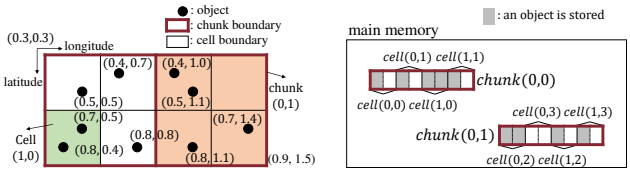
*Definition 2.2 (**Chunk**).* Suppose that (1) the number of cells along latitude in a single chunk is $\boldsymbol{nCell_{lat}^{chunk}}$, and (2) the number of cells along longitude in a single chunk is $\boldsymbol{nCell_{lon}^{chunk}}$. The cell coordinate of a cell, $(cell_{lat}, cell_{lon})$, is mapped to the integer **chunk coordinate**, $(chunk_{lat}, chunk_{lon})$, where

$$chunk_{lat} = \lfloor \frac{cell_{lat}}{nCell_{lat}^{chunk}} \rfloor$$
$$chunk_{lon} = \lfloor \frac{cell_{lon}}{nCell_{lon}^{chunk}} \rfloor. \quad (2)$$

A **chunk**, $chunk(chunk_{lat}, chunk_{lon})$, is a set of all possible cells, of which (1) chunk coordinates are $(chunk_{lat}, chunk_{lon})$, and (2) cell coordinates are unique in the set. A chunk also includes empty cells if they exist. □

In Waffle, a chunk is the unit of serial memory allocation in the main memory. The neighboring cells that construct a chunk tend to be accessed consecutively during query processing, and the access is likely to be cache-friendly through serial allocation. Without loss of generality, Waffle assumes that the cells in a chunk are stored in the latitude-major order of the cell coordinates. A chunk consisting of only empty cells is called an empty chunk. If Waffle requires more than one cell with the same cell coordinate to store the objects, Waffle creates more than one chunk because the cell coordinates are unique in a chunk. We describe the detailed query processing in Section 3. $nCell_{lat}^{chunk}$ and $nCell_{lon}^{chunk}$ are the knobs of Waffle used to define chunks.

We call a set of five knob values, $\{nCell_{lat}^{space}, nCell_{lon}^{space}, MOPC, nCell_{lat}^{chunk}, nCell_{lon}^{chunk}\}$, a **knob setting**. Determination of a knob setting has a huge impact on the overall performance of Waffle, and we describe a method for determining a knob setting in Section 6. If $nCell_{lat}^{\{space/chunk\}} = nCell_{lon}^{\{space/chunk\}}$, we briefly denote the two knobs as $nCell^{\{space/chunk\}}$.



(a) Cells and chunks **(b) Memory representation**

Figure 1: Cell and chunk definition

Figure 1 shows an example Waffle grid index, where the space is $([0.3, 0.9), [0.3, 1.5))$, and the knob setting is $\{2, 4, 2, 2, 2\}$. Each object is uniquely mapped to a cell, and each cell is uniquely mapped to a chunk. Figure 1(b) shows that the cells of each chunk are serially placed in the main memory, but the two chunks are scattered.

3 QUERY PROCESSING

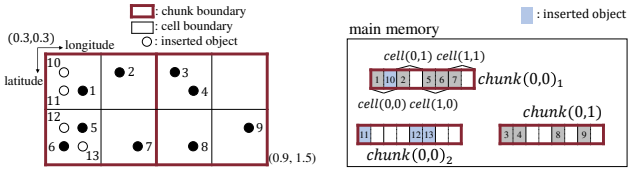
Waffle supports four query types: insertion, deletion, range, and k-NN queries [1]. We introduce query processing based on the cell and chunk structure.

(1) Insertion query: Given the key of an object and its coordinate, an insertion query adds the object to a Waffle index. We introduce query processing based on the example in Figure 2. The objects in the figure are inserted into a Waffle index with the same knob setting as Figure 1 in an ascending order of their keys. The cell and chunk coordinate of Object 10 are both $(0, 0)$, and the object is inserted into $cell(0, 0)$ in $chunk(0, 0)_1$. The cell and chunk coordinate of Object 11 are also $(0, 0)$. Waffle first checks whether $cell(0, 0)$ in $chunk(0, 0)_1$ has less than $MOPC(= 2)$ objects. However, because the cell already has $MOPC$ objects, Waffle creates a new chunk with the same chunk coordinate, $chunk(0, 0)_2$, and inserts Object 11 into the chunk. The cell and chunk coordinate of Object 12 are $(1, 0)$ and $(0, 0)$, respectively. Waffle checks whether $cell(1, 0)$ in $chunk(0, 0)_1$ has already $MOPC$ objects. Waffle then moves to the next chunk, $chunk(0, 0)_2$, created during the insertion query for Object 11 and inserts Object 12 into the chunk. Objects 13 is inserted in the same manner.

(2) Deletion query: Given the key of an object, a deletion query removes the object from a Waffle index. We introduce query processing based on the example in Figure 3 starting from the grid index in Figure 2. First, Waffle deletes Object 1. The cell and chunk coordinate of Object 1 are $(0, 0)$. Waffle checks $chunk(0, 0)_1$ and

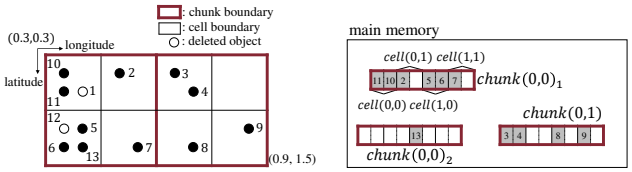
[1] Even if insertion and deletion are not queries but operations, this paper calls them queries for the sake of brevity.

finds Object 1. Then, Waffle finds the last object of the cell coordinate, Object 11, swaps Object 1 with Object 11, and marks Object 1 as deleted. Next, Waffle deletes Object 12. The cell and chunk coordinate of Object 12 are $(1, 0)$ and $(0, 0)$, respectively. Waffle first checks Object 5 and then Object 6 in $chunk(0, 0)_1$ and fails to find Object 12. Waffle moves to the next chunk, $chunk(0, 0)_2$, swaps Object 12 with Object 13, and marks Object 12 as deleted. If Object 13 is also deleted, Waffle deletes $chunk(0, 0)_2$ because it becomes an empty chunk.



(a) Cells and chunks **(b) Memory representation**

Figure 2: Example of insertion query processing



(a) Cells and chunks **(b) Memory representation**

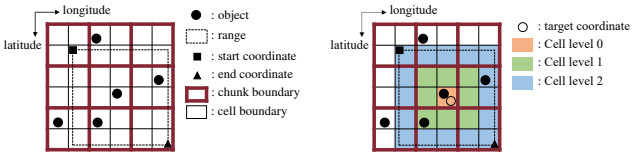
Figure 3: Example of deletion query processing

(3) Range query: A range query returns all objects within a rectangular range specified by a start and end coordinate. Waffle processes a range query as follows. If a chunk is completely within the range boundary, all objects in the chunk are added to the answers. If a chunk overlaps with the range boundary, a cell in the chunk is completely within or overlaps with the range boundary. In the former case, all objects in the cell are added to the answers. In the latter case, Waffle checks whether each object in the cell is within the range boundary.

(4) k-NN query: Given a target coordinate, k, and a rectangular range specified by a start and end coordinate, a k-NN query returns k objects closest to the target coordinate within the range. We adopt the k-NN query processing in the grid index from [46]. First, we define a cell level. Assume that the coordinate of the cell containing the target coordinate is (t_{lat}, t_{lon}). Cell level 0 is defined as $\{cell(t_{lat}, t_{lon})\}$. Cell level i ($\geq$ 1) is defined as $\{cell(cell_{lat}, cell_{lon}) \mid t_{lat} - i \leq cell_{lat} \leq t_{lat} + i, t_{lon} - i \leq cell_{lon} \leq t_{lon} + i\} - \bigcup_{j=0}^{i-1}$Cell level j. The example cell levels are shown in Figure 4(b).

Waffle maintains a priority queue that contains three types of elements: an object, a non-empty cell, and a cell level. For an element, the Euclidean distance from the target coordinate is calculated. An element closer to the target coordinate has a higher priority in the queue. Specifically, for a cell, the shortest distance between the cell boundary and the target coordinate is calculated. For a cell level, the distance from the closest cell at the level is calculated.

Waffle only considers elements overlapping with or included in the range boundary. First, Waffle inserts (1) all objects at Cell level 0 and (2) Cell level 1 into the priority queue. If the top of the queue is an object, it becomes a new answer. If the top of the queue is a cell, all objects in the cell are inserted into the queue. If the top of the queue is a cell level, (1) all non-empty cells at the cell level and (2) the next cell level are inserted into the queue. If k objects are retrieved, or the queue becomes empty, then the query processing is completed.



(a) Range query **(b) k-NN query**

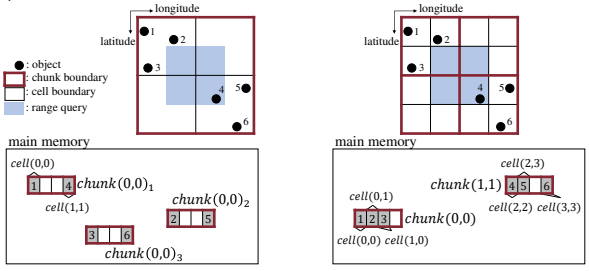
Figure 4: Example of range and k-NN query processing

4 EFFECTS OF KNOB CONFIGURATION

Section 2 introduced the cell and chunk definition based on the five knobs. We next describe detailed configuration effects from the perspectives of query processing and memory usage using examples. Assume that the objects in the examples are inserted in an ascending order of their keys and then deleted in a decreasing order of their keys, and the range queries are processed right after all the objects are inserted.

4.1 Effects of the Number of Cells in the Space

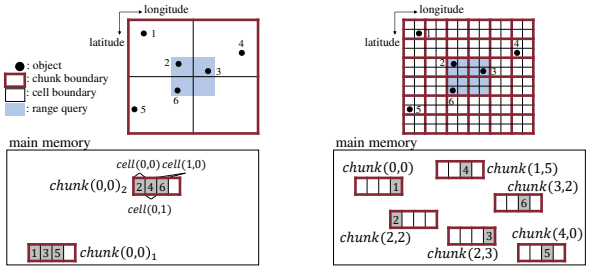
Figure 5 shows an example when $MOPC = 1$ and $nCell^{chunk} = 2$. When Object 5 is inserted, in Figure 5(a), Waffle first checks $chunk(0, 0)_1$ to check whether $cell(1, 1)$ in the chunk already contains $MOPC$ object. Then, Waffle inserts the object into the next chunk, $chunk(0, 0)_2$. In Figure 5(b), Object 5 is inserted into $chunk(1, 1)$ without checking additional chunks. When Object 6 is deleted, in Figure 5(a), Waffle checks Object 4 in $chunk(0, 0)_1$ and Object 5 in $chunk(0, 0)_2$ and then finds Object 6 in $chunk(0, 0)_3$. In Figure 5(b), Object 6 is deleted from $chunk(1, 1)$ without checking additional objects. For the range query in the figures, in Figure 5(a), Waffle checks all the objects because the range boundary overlaps with all the cells. In figure 5(b), Waffle checks only the single object. Figure 5(a) is likely to have more cache misses than Figure 5(b) because the former checks more chunks. From the memory usage perspective, Figure 5(b) results in less memory usage than Figure 5(a).



(a) $nCell^{space} = 2$ **(b) $nCell^{space} = 4$**

Figure 5: Example effects of $nCell^{space}$

Figure 6 shows another example when $MOPC = 1$ and $nCell^{chunk} = 2$. When Object 6 is inserted, in Figure 6(a), Waffle first checks $chunk(0, 0)_1$ and then inserts the object into the next chunk, $chunk(0, 0)_2$. In Figure 6(b), Object 6 is inserted into $chunk(3, 2)$ without checking additional chunks. When Object 4 is deleted, in Figure 6(a), Waffle checks Object 3 in $chunk(0, 0)_1$



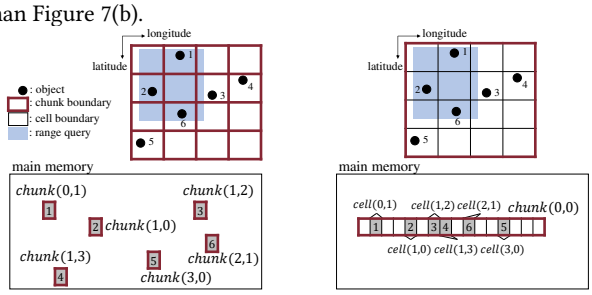
(a) $nCell^{space} = 2$ (b) $nCell^{space} = 12$

Figure 6: Another example effects of $nCell^{space}$

and then finds Object 4 in $chunk(0,0)_2$. In Figure 6(b), Object 4 is deleted from $chunk(1,5)$ without checking additional objects. For the range query in the figures, in Figure 6(a), Waffle checks all the objects because the range boundary overlaps with all the four cells. In Figure 6(b), Waffle checks only three objects, but they are scattered in the main memory. Figure 6(b) is likely to have more cache misses than Figure 6(a). From the memory usage perspective, Figure 6(a) results in less memory usage than Figure 6(b).

4.2 Effects of the Number of Cells in a Chunk

Figure 7 shows an example when $nCell^{space} = 4$ and $MOPC = 1$. For the insertion queries, in Figures 7(a)-7(b), all the objects are inserted without checking additional chunks. For the deletion queries, in both figures, all the objects are deleted without checking additional objects. For the range query, the number of checked objects during query processing is the same in both figures. Instead, for an insertion/deletion/range/k-NN query, Figure 7(b) is more cache-friendly than Figure 7(a) because of the serial memory allocation. From the memory usage perspective, Figure 7(a) uses less memory than Figure 7(b).

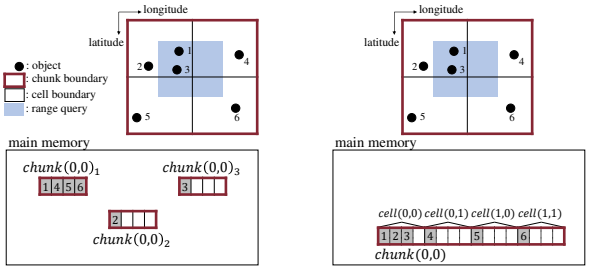


(a) $nCell^{chunk} = 1$ (b) $nCell^{chunk} = 4$

Figure 7: Example effects of $nCell^{chunk}$

4.3 Effects of the Maximum Objects per Cell

Figure 8 shows an example when $nCell^{space} = nCell^{chunk} = 2$. When Object 3 is inserted, in Figure 8(a), Waffle first checks $chunk(0,0)_1$ and $chunk(0,0)_2$ and then inserts the object into $chunk(0,0)_3$. In Figure 8(b), Object 3 is inserted into $chunk(0,0)$ without checking additional chunks. When Object 3 is deleted, in Figure 8(a), Waffle first checks Object 1 and Object 2 and then finds Object 3 in $chunk(0,0)_3$. In Figure 8(b), Waffle also checks Object 1 and Object 2 and then finds Object 3 in $chunk(0,0)$. For the range query, Figure 8(a) is cache-friendly when reading the neighboring cells in each chunk. Figure 8(b) is cache-friendly when reading the objects with the same cell coordinate. From the memory usage perspective, Figure 8(a) uses less memory than Figure 8(b).



(a) $MOPC = 1$ (b) $MOPC = 4$

Figure 8: Example effects of $MOPC$

4.4 Discussion

For an insertion/deletion query, the number of checked chunks/objects affects computational cost and cache misses. After an insertion/deletion query is processed, if Waffle processes the next insertion/deletion query while minimizing cache misses, the query processing becomes efficient. For a range/k-NN query, if checked objects are serially placed in the main memory, and non-answer objects are checked less, then the query processing becomes more efficient. If the knob values are smaller, Waffle tends to use less memory.

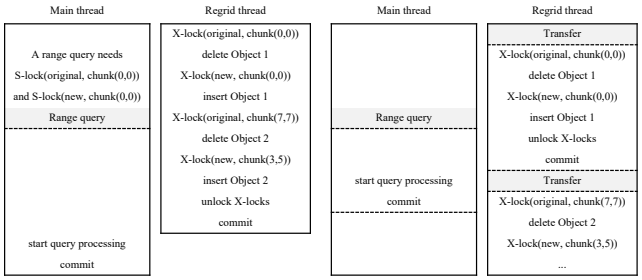
However, generalization and formulation of the configuration effects are not trivial. The example configuration effects are only a few possible scenarios, and the effects may change if the other knob values are different. An appropriate configuration is affected by various factors including object distribution, query types and workload, hardware configuration, and a trade-off between query processing time and memory usage. Furthermore, a configuration has a significant impact on the performance of Waffle. The difficulty of generalization and the importance of an appropriate configuration become the motivations of automatic configuration tuning.

5 REGRID: A MECHANISM FOR REDEFINING A WAFFLE INDEX

Section 4 discussed the several effects of knob configuration through the examples. Before introducing how to automatically determine a knob setting in Section 6, we propose redefining a Waffle index.

Whenever x queries are processed, given a new knob setting, Waffle reorganizes a grid index, called a regrid. Specifically, for each object in the original grid index, Waffle deletes it from the original index and inserts it into the new grid index. When no objects remain in the original index, the new index replaces the original index, which means that the regrid is completed.

While rebuilding a Waffle index, without a mechanism to handle user queries, the query processing is blocked, and the blocked time increases when the number of objects is large. To overcome this problem, Waffle includes a concurrency control scheme through a strict two-phase locking protocol. Waffle sets the lock granularity as a set of chunks with the same chunk coordinate because a chunk is the unit of serial memory allocation, and the chunks with the same chunk coordinate are likely to be accessed together during query processing. An insertion/deletion query requires an exclusive lock (X-lock), and a range/k-NN query requires a set of shared locks (S-lock). When a transaction requires a set of locks, a deadlock problem is prevented by setting a partial order as follows. (1) Locks



(a) Original regrid **(b) Split regrid**

Figure 9: Example schedules for a regrid

in the original index are obtained earlier than locks in the new index. (2) In an index, for $chunk_1 = chunk(lat_1, lon_1)$ and $chunk_2 = chunk(lat_2, lon_2)$, if $lat_1 < lat_2$, the lock for $chunk_1$ is obtained earlier than that for $chunk_2$. If $lat_1 = lat_2$ and $lon_1 < lon_2$, the lock for $chunk_1$ is obtained earlier than that for $chunk_2$.

Based on the locking protocol, Waffle splits a regrid into a set of **transfer** transactions, each of which consists of one deletion query for an object from the original index and one insertion query for the object into the new index. Waffle processes concurrent user queries without waiting until all transfer transactions from the regrid are completed. Figure 9 shows the example schedules for a regrid. While a range query from a user cannot be processed until the regrid is completed if not split (Figure 9(a)), Waffle processes the range query after one transfer transaction (Figure 9(b)).

Splitting a regrid based on the concurrency control scheme requires modification of the query processing during a regrid. Without loss of generality, we assume that (1) a user corresponds to a unique object, (2) if the user issues a query, the user is prevented from issuing other queries before the query processing is completed, and (3) user queries from multiple users are processed in a single thread, and a regrid is processed in another thread [2]. The detailed query processing is as follows.

(1) User insertion query: If a given object does not exist in both the original and new indexes, Waffle processes a transaction for an insertion query to the new index without considering the original index. If the given object already exists in the original or new index, Waffle processes the request as a movement query, as described later.

(2) User deletion query: If an object is still in the original index, Waffle processes a transaction for a deletion query from the original index. If the object is in the new index, Waffle processes a transaction for a deletion query from the new index.

(3) User movement query: In Section 3, we do not discuss a movement query of an object because it can be simply replaced with a deletion query and a subsequent insertion query without considering the concurrency [3]. With the concurrency control scheme, Waffle should consider a movement query separately; that is, Waffle processes a transaction for (1) a deletion query and (2) a subsequent insertion query. If the deletion and insertion query for an object are not processed as a transaction, after the deletion query is processed, another concurrent transaction cannot determine whether the object has been deleted, or the movement query is being processed. If the target object is still in the original index, Waffle processes a transaction for (1) a deletion query from the original index and (2) a subsequent insertion query into the new index. If the target object is in the new index, Waffle processes a transaction for (1) a deletion query from the new index and (2) a subsequent insertion query into the new index.

(4) Transfer transaction: Before processing a transfer transaction for an object, if a user insertion/deletion/movement query targeting the same object is being processed, Waffle waits until the query processing is completed. After the user query finishes, Waffle starts the transfer transaction and forces another user insertion/deletion/movement query targeting the same object to wait.

For each object still in the original index, Waffle processes a transfer transaction for (1) a deletion query from the original index and (2) a subsequent insertion query into the new index. Waffle does not have to transfer an object already deleted from the original index because the previous user deletion or movement query deleted the object from the space or updated the coordinate of the object in the new index.

(5) User range/k-NN query: Waffle processes a transaction for two range/k-NN queries: a range/k-NN query for the original index and a subsequent range/k-NN query for the new index. Before processing each query, the transaction obtains all the required shared locks for the target index. The correct answers are constructed from the union of the two range/k-NN answers, and the reasons are as follows. (1) When another transaction A holds an exclusive lock for updating the chunk that is also required by the range/k-NN transaction, the range/k-NN transaction should wait until A releases the lock and the range/k-NN transaction obtains the shared lock. (2) When the range/k-NN transaction holds the shared lock for a chunk, another transaction B that updates the same chunk should wait until the range/k-NN transaction releases the lock and B obtains the exclusive lock. (3) The movement/transfer processing ensures that an object in the geographical space exists only in either the original index or the new index.

6 WaffleMaker: REINFORCEMENT LEARNING -BASED CONFIGURATION TUNING SYSTEM

In Section 2, we introduced the five knobs to define cells and chunks. Section 4 described the various configuration effects. In Section 5, given a new knob setting, we proposed a novel regrid process based on a concurrency control scheme. In this section, we propose an online configuration tuning system, WaffleMaker, for determining the knob setting for a regrid. WaffleMaker is based on reinforcement learning performing a novel type of exploration and exploitation and does not require pre-training. Section 6.1 introduces the basic components of WaffleMaker. Section 6.2 defines the model, and Section 6.3 describes the workflow of WaffleMaker. Finally, Section 6.4 discusses the characteristics of WaffleMaker from several perspectives.

6.1 Basic Components

We first define the basic components of WaffleMaker: a state, an action, a reward, and an experience.

[2] If user queries are processed in a massively parallel environment, different concurrency control schemes including latch-free methods may be considered. However, this is out of the main focus of this paper.

[3] A movement query within a cell boundary can be optimized. However, we do not consider the case in this paper.

(1) State (S): A WaffleMaker state is defined as a grid summarizing the current objects within a geographical space. Specifically, given (1) a geographical space, (2) the number of state cells along latitude $nCell_{lat}^{state}$, and (3) the number of state cells along longitude $nCell_{lon}^{state}$, a WaffleMaker state is a $nCell_{lat}^{state} \times nCell_{lon}^{state}$-sized grid, each cell of which maintains the number of corresponding objects. Note that a state grid is not related to a Waffle grid index. $nCell_{lat}^{state}$ and $nCell_{lon}^{state}$ are the hyperparameters of WaffleMaker, which means that they do not change during the runtime of Waffle.

(2) Action: A WaffleMaker action is to decide a knob setting KS. Specifically, each knob is matched with a single real-number variable representing a value in $[0, 1]$, which means that action space is continuous. Given the minimum and maximum value of each knob, the value of the variable is mapped to a knob value. The continuous action space enables WaffleMaker to represent any range of a knob with a single variable.

(3) Reward (R): A WaffleMaker reward represents the quality of an action. A reward consists of five elements: insertion, deletion, range, k-NN, and memory reward.

{Insertion/Deletion/Range/k-NN} reward ($R_{\{i/d/r/k\}}$): Suppose that $x_{\{i/d/r/k\}} (\neq 0)$ {insertion/deletion/range/k-NN} queries are processed after the regrid with KS has been completed. The {insertion/deletion/range/k-NN} reward for the action is the negative value of the average {insertion/deletion/range/k-NN} time for the $x_{\{i/d/r/k\}}$ queries.

$$R_{\{i/d/r/k\}} = -\frac{total\ execution\ time\ for\ x_{\{i/d/r/k\}}\ queries}{x_{\{i/d/r/k\}}} \quad (3)$$

We reverse the sign to provide a higher reward to a shorter execution time.

Memory reward (R_m): Suppose that the total number of chunks is $|chunks|$, and memory usage of a single chunk is $mChunk$. The memory reward is the negative value of approximate memory usage.

$$R_m = -|chunks| \times mChunk \quad (4)$$

$|chunks| \times mChunk$ approximates total memory usage of a Waffle index at a specific moment because a single chunk is the memory allocation unit, and memory usage of a single chunk is constant regardless of the existence of objects in the chunk if the knob setting does not change. $|chunks|$ may change from insertion and deletion queries; WaffleMaker obtains the values at the moment the memory reward is measured.

Before calculating the final reward, each element is normalized to prevent a particular element from being excessively reflected to the final reward. We describe a method for normalizing each element in Section 7.2.2. Suppose that (1) the normalized rewards of insertion, deletion, range, k-NN, and memory are R'_i, R'_d, R'_r, R'_k, and R'_m, respectively, and (2) the weights for query processing time and memory usage are w_{time} and w_{memory}, respectively, where $w_{time} + w_{memory} = 1$. The final reward R is defined as follows.

$$R = w_{time}\{\frac{R'_i + R'_d + R'_r + R'_k}{4}\} + w_{memory}R'_m \quad (5)$$

The weight values are the hyperparameters of WaffleMaker.

(4) Experience: Assume that the current state is S, and WaffleMaker determines a new knob setting KS and obtains the reward R for the action. An experience is defined as (S, KS, R).

6.2 WaffleMaker Model

Given a state S and a knob setting KS, the WaffleMaker model outputs an expected reward, which means that the model is an action-value function, $Q(S, KS) = R$. The model is a convolutional neural network that consists of convolutional layers, pooling layers, and fully connected layers. A convolutional neural network enables sparse interactions considering spatial locality [11]. In addition, the model learns from the state without requiring additional information. This insight is similar to that of the previous studies [26, 37]. For example, the study [26] proposed a deep reinforcement learning model with a convolutional neural network that learns how to play Atari games directly from the pixel values.

The model is updated in an off-policy manner. Given a set of experiences (S, KS, R), WaffleMaker samples a mini-batch of $|batch|$ experiences from the set and updates the model according to the following mean squared error.

$$loss = \frac{1}{|batch|}\sum_{i=1}^{|batch|}(Q(S_i, KS_i)) - R_i)^2 \quad (6)$$

Given a state S, WaffleMaker determines a knob setting KS as follows. For **exploration**, WaffleMaker randomly selects $|candidates|$ knob settings. WaffleMaker then passes the knob settings as a single mini-batch to the model, obtains the expected rewards, and sorts the knob settings in a descending order of the expected rewards. For the top-T% knob settings with regard to the expected rewards, WaffleMaker samples a knob setting considering the expected rewards as sampling weights. As the model becomes more accurate, an actual final reward from exploration is expected to be higher.

For **exploitation**, WaffleMaker utilizes a set of $|recent|$ knob settings determined by recent explorations. Specifically, WaffleMaker passes the recent knob settings as a mini-batch to the model and obtains the top-1 knob setting with the highest expected reward. Determining a knob setting from the recent knob settings means that continuous action space is replaced with discrete action space, where the discrete actions are not fixed but changing. As the model becomes more accurate, the current set of recent knob settings is expected to lead to higher final rewards than the previous set. The novel exploitation method enables Waffle to perform a regrid in a timely manner, specifically about determining when to perform a regrid.

$|batch|$, $|candidates|$, T, and $|recent|$ are the hyperparameters of WaffleMaker. As the model becomes more accurate, gradually reducing T is an option.

6.3 Workflow of WaffleMaker

We introduce the workflow of WaffleMaker based on Figures 10-11. Waffle starts a new regrid whenever x queries are processed after the previous regrid finishes. The workflow varies depending on whether the model converges.

We first explain the workflow before the model converges with Figure 10. Waffle processes x queries and starts a new regrid (Steps 1-2). WaffleMaker decides a new knob setting KS from the current state S (Step 3). WaffleMaker obtains the knob setting through exploration with probability $P_{exploration}$ and exploitation with probability $1 - P_{exploration}$. $P_{exploration}$ is the hyperparameter of WaffleMaker.

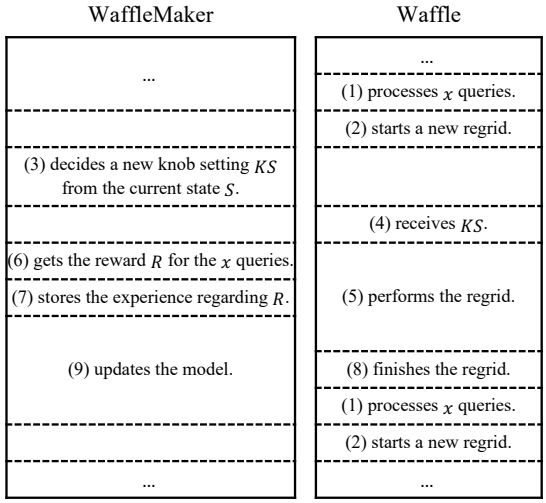


Figure 10: Workflow of WaffleMaker (before convergence)

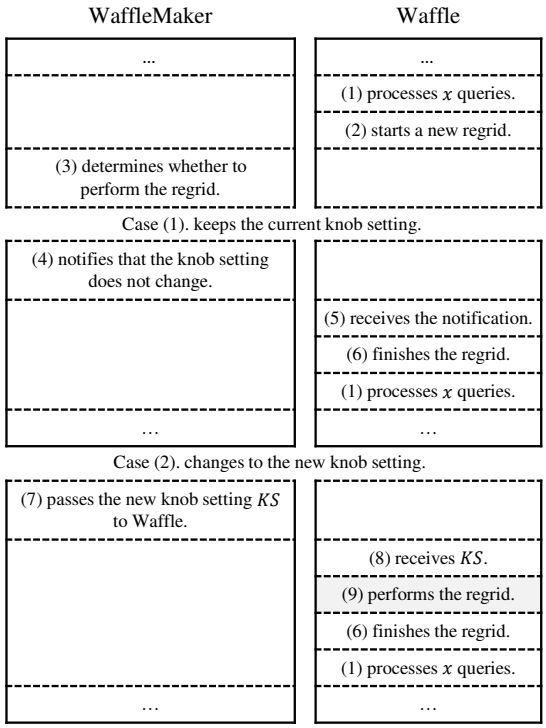


Figure 11: Workflow of WaffleMaker (after convergence)

After WaffleMaker decides the new knob setting, Waffle receives the knob setting from WaffleMaker (Step 4). Then, Waffle performs the regrid while processing the concurrent user queries, as described in Section 5 (Step 5).

While Waffle performs the regrid, WaffleMaker calculates the reward R for the x queries based on Equation 5 (Step 6), stores the new experience regarding R into an experience buffer (Step 7), and updates the model utilizing the buffer (Step 9), as detailed in Section 6.2.

Waffle finishes the regrid (Step 8) and processes subsequent x queries (Step 1). The x queries do not include user queries processed during the previous regrid because query processing during the regrid is affected by two different knob settings. Waffle starts a new regrid (Step 2), and the process is repeated.

The workflow after the model converges is shown in Figure 11. Waffle processes x queries and starts a new regrid (Steps 1-2). Then, WaffleMaker determines whether to perform the regrid (Step 3). Specifically, WaffleMaker obtains a candidate knob setting from the current state through exploitation. If the candidate knob setting is the same as the current knob setting, the current one still has the highest expected reward for the current state among the $|recent|$ knob settings, and WaffleMaker determines not to perform the regrid (Case 1). Otherwise, the current knob setting no longer has the highest expected reward for the current state among the $|recent|$ knob settings, and WaffleMaker determines to perform the regrid (Case 2).

6.4 Discussion

(1) Insight: The two primary motivations of WaffleMaker are (1) how to determine a knob setting and (2) when to perform a regrid. For the first motivation, the process to determine a knob setting is modeled as a reinforcement learning problem, where a state is defined from object distribution, an action is to determine a knob setting, and a reward is calculated from the index performance. For the second motivation, instead of building additional models or rules, WaffleMaker naturally determines when to perform a regrid by replacing continuous action space with discrete action space. After convergence, WaffleMaker periodically determines a candidate knob setting from the current state and checks whether the candidate is different from the existing one. If the two knob settings are the same, WaffleMaker recognizes that the object distribution has not changed enough to perform the regrid. Otherwise, WaffleMaker perceives that the object distribution has changed enough. In summary, WaffleMaker integrates the two motivations.

(2) Contextual k-armed bandit problem: WaffleMaker defines an automatic configuration tuning process as a contextual k-armed bandit problem [1, 14, 16, 18, 20, 24, 25, 38], which is under the category of reinforcement learning. The main characteristic of a contextual bandit problem is that an action does not affect the next state (i.e., context), and therefore, the reward is not delayed but immediate. In WaffleMaker, an action does not affect the next state because the state is a fixed-sized grid just representing an object distribution. A state changes not by a knob setting (i.e., action) but by user insertion/deletion/movement queries, and the user queries are not under the control of WaffleMaker. A different type of action and state affected by previous actions may be defined for a Markov decision process. Instead, WaffleMaker models the natural process of observing the current object distribution, determining a knob setting, and obtaining index performance, as a contextual bandit problem.

However, it is difficult to directly apply the existing contextual k-armed bandit methods to our configuration tuning problem. As an option, all possible knob settings (i.e., k arms) are enumerated, and one of the existing methods is applied, which results in excessively large action candidates. As another option, a bandit model may be built for each knob. However, a reward derives from a combination of the five knob values, which makes it ambiguous how to update each bandit model.

7 EXPERIMENTAL EVALUATION

7.1 Experimental Setup

Settings. We conducted our experiments using a server with an Intel(R) Xeon(R) Silver 4215R CPU (3.20 GHz), 256 GB memory, a 2 TB NVMe SSD, and an NVIDIA Quadro RTX 8000. The L1 data cache size is 32 KB, L1 instruction cache size is 32 KB, L2 cache size is 1024 KB, L3 cache size is 11 MB, and cache line size is 64 B. The server was operated by Ubuntu 18.04.5 LTS. All the evaluated systems and algorithms were implemented using C++14 and PyTorch 1.8.1 with the PyTorch C++ frontend if required. All the results in the graphs and tables were averaged from 10 executions.

Evaluated Systems and Algorithms.

(1) Waffle: Waffle consists of five main components: grid index manager, lock manager, transaction manager, regrid manager, and WaffleMaker. The hyperparameters of WaffleMaker used in the experiments were as follows: $nCell_{lat}^{state} = nCell_{lon}^{state} = 128$, $w_{time} = 0.9$, $w_{memory} = 0.1$, $|batch| = 64$, $|candidates| = 1000$, $T = 5$, $|recent| = 200$, and $P_{exploration} = 0.8$. The action-value function consisted of four convolutional layers and three fully connected layers. The learning rate was 10^{-3}. The hyperparameters common to DDPG including $|batch|$, the network structure, and the learning rate were tuned based on DDPG, which is described below. We implemented a prioritized experience replay [33]. For an experience (S, KS, R), a priority was set to $(Q(S, KS)) - R)^2$.

(2) DDPG [23]: A policy gradient method, which has been widely adopted for automatic database configuration tuning [3, 19, 44]. DDPG was a baseline for WaffleMaker. DDPG supported continuous action space and could be easily modified to consider an immediate reward, instead of a delayed reward, as discussed in Section 6.4. DDPG consisted of two models: an actor and a critic. For exploration, DDPG added a noise to an action determined by the actor, where the noise came from an Ornstein-Uhlenbeck process [41] with $\theta = 0.15$ and $\sigma = 0.15$. For exploitation, DDPG utilized an action determined by the actor without adding a noise. The learning rate of the actor was 10^{-4}, and that of the critic was 10^{-3}. We also implemented a prioritized experience replay [33]. Because we considered only an immediate reward, a priority was replaced with $|critic(S, KS) - R|$.

(3) u-Grid [35] and **(4) u-R-tree** [35]: An in-memory grid and R-tree index optimized for moving objects, which was a baseline for Waffle.

(5) Quad tree [10]: A multiple resolution grid index implemented at in-memory setting, which was a baseline for Waffle.

(6) RSMI [29]: A recursive model index [15] for spatial data, which was a baseline for Waffle. We utilized the source code uploaded on the GitHub website of the author, implemented at in-memory setting. N and B were set to 20000 and 200, respectively.

All the methods processed a series of queries in a single thread, and Waffle executed up to two additional threads for WaffleMaker and a regrid only if required.

Datasets. We used the road network datasets [9], and each geographical space was as follows.

(1) LA: ([33.8449, 34.3243), [−118.766, −117.793))

(2) NewYork: ([40.2513, 41.0595), [−74.8782, −73.2227))

An **episode** in the experiments was defined as follows. $|base|$ objects were placed on random roads or at the last positions of the previous episode and kept randomly moving along roads. $|extra|$ objects were gradually inserted into roads around the center of the geographical space and kept randomly moving along roads. Then, the $|extra|$ objects were gradually deleted from the space while the $|base|$ objects were still randomly moving.

In the middle of insertion and deletion queries, a range or k-NN query was created alternatively by setting the coordinate of a random object as the center of its range. The range size of a range/k-NN query was randomly set to [0.5%, 1.5%] of the geographical space size of each dataset. For a k-NN query, k was set to 10, and the target coordinate was set to the center of its range.

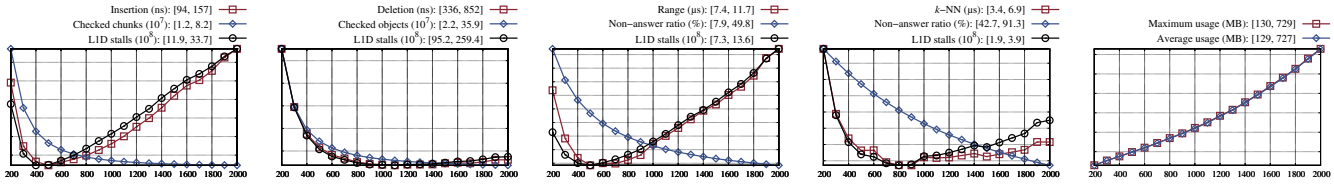
$|base|$ and $|extra|$ were both set to 10^6 as a default setting. For the values, the approximate number of insertion/deletion queries during an episode was 1.7×10^7, and that of range/k-NN queries was 9×10^4. Before each experiment, we generated queries and stored them on the storage. A method read the queries one by one from the storage, and the reading time was excluded from an execution time.

7.2 Experimental Results

7.2.1 Experiments for Configuration Effects. Figures 12-14 show the results according to various knob settings to check the configuration effects described in Section 4 for LA dataset. We set $|base|$ to 10^6 and did not consider $|extra|$ objects in the experiments. We recorded the average query execution time for each query type, memory usage during the execution, and additional information. A non-answer ratio was calculated from $(1 - \frac{|answers|}{|checked\ objects|}) \times 100$ for a range/k-NN query. We also measured execution stalls from L1 data cache misses using perf, which is a performance analysis tool in Linux.

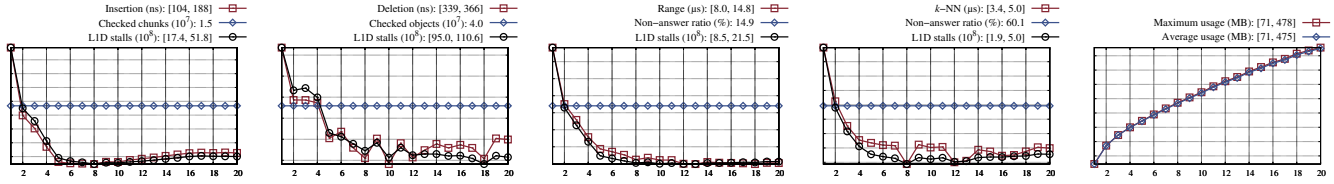
First, the results according to the number of cells in the space ($nCell^{space}$) are shown in Figure 12. In Figure 12(a), when $nCell^{space}$ was small, the number of checked chunks increased because Waffle created more cells with the same cell coordinate, as shown in Figures 5(a) and 6(a). As the number of checked chunks increased, the computational costs and stalls also increased, which resulted in longer insertion time. When $nCell^{space}$ was large, even if the number of checked chunks decreased, the stalls increased because each insertion was likely to cause more cache misses, as shown in Figure 6(b). Figure 12(b) shows the deletion times. Unlike an insertion query, a deletion query checks other objects with the same cell coordinate. When $nCell^{space}$ was small, the number of checked objects increased because more objects had the same cell coordinate, as shown in Figures 5(a) and 6(a). Accordingly, the computational costs and stalls also increased, and the deletion time also increased.

In Figure 12(c), when $nCell^{space}$ was small, more computational costs occurred from checking more non-answer objects, as shown in Figures 5(a) and 6(a). When $nCell^{space}$ was large, even if the non-answer ratio decreased, more stalls occurred from checking more chunks, as shown in Figure 6(b), and the range query time increased. Figure 12(d) shows the results for k-NN queries, each of which required a priority queue. When $nCell^{space}$ was small, the larger number of checked objects caused more computational costs and stalls. When $nCell^{space}$ was large, even if the non-answer ratio decreased, accessing objects was likely to cause more stalls. Figure 12(e) shows that Waffle used more memory when $nCell^{space}$ increased, as shown in Figure 6(b).



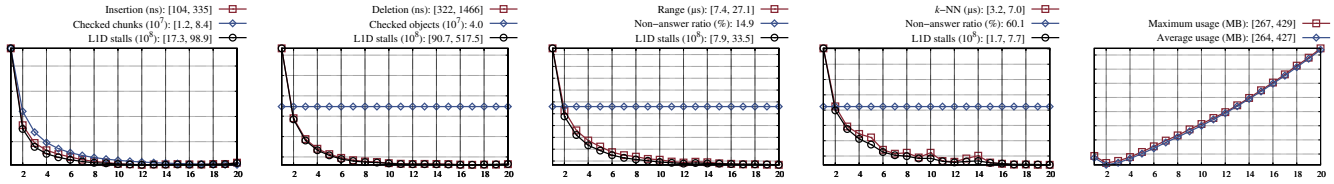
(a) Insertion (b) Deletion (c) Range (d) k-NN (e) Memory usage

Figure 12: Results according to $nCell^{space}$ ($MOPC = 10$, $nCell^{chunk} = 10$, LA)



(a) Insertion (b) Deletion (c) Range (d) k-NN (e) Memory usage

Figure 13: Results according to $nCell^{chunk}$ ($nCell^{space} = 1000$, $MOPC = 10$, LA)



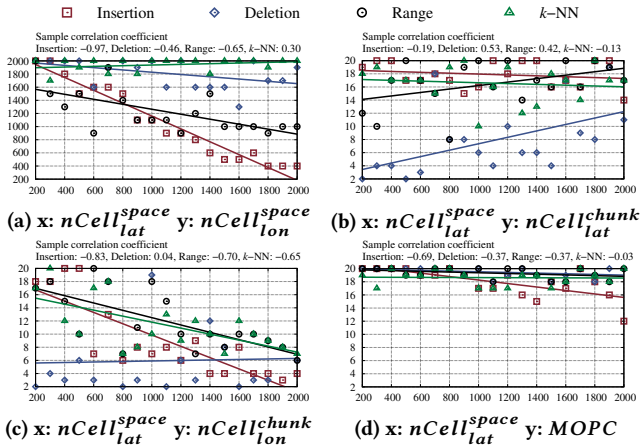
(a) Insertion (b) Deletion (c) Range (d) k-NN (e) Memory usage

Figure 14: Results according to $MOPC$ ($nCell^{space} = 1000$, $nCell^{chunk} = 10$, LA)

Figure 13 shows the results according to the number of cells in a chunk ($nCell^{chunk}$). In Figure 13(a), the number of checked chunks remained the same because $nCell^{space}$ and $MOPC$ were fixed, as shown in Figure 7, which means that the overheads from computational costs were similar. However, when $nCell^{chunk}$ increased, the stalls tended to decrease, as shown in Figure 7(b), and the insertion time also tended to decrease. In Figure 13(b), unlike in Figure 12(b), the number of checked objects was the same because $nCell^{space}$ was fixed, which resulted in similar computational costs. However, the larger $nCell^{chunk}$ tended to reduce the stalls, and the deletion time tended to decrease. Figures 13(c)-13(d) show that Waffle tended to process the range and k-NN queries faster when $nCell^{chunk}$ was large because of the fewer stalls. In addition, the non-answer ratio remained the same because of the fixed $nCell^{space}$, as shown in Figure 7. Figure 13(e) shows that Waffle tended to require more memory when $nCell^{chunk}$ increased, as shown in Figure 7(b).

Figure 14 shows the results according to the maximum objects per cell ($MOPC$). In Figure 14(a), when $MOPC$ increased, the number of checked chunks and stalls tended to decrease, as shown in Figure 8(b). In Figure 14(b), $MOPC$ did not affect the number of checked objects, as shown in Figure 8. However, when $MOPC$ was small, more chunks were created, and Waffle checked the same objects at different chunks, which resulted in more stalls and inefficient deletion query processing. In Figures 14(c)-14(d), when $MOPC$ increased, the range and k-NN query processing became efficient because of the fewer stalls. Figure 14(e) shows that Waffle tended to occupy more memory when $MOPC$ increased.

Figures 12-14 show that the performance of Waffle varies significantly according to the knob configurations. The results shown in the graphs may be different if the default knob values, object distribution, and user queries change. The results support the necessity of WaffleMaker automatically tuning knob values.

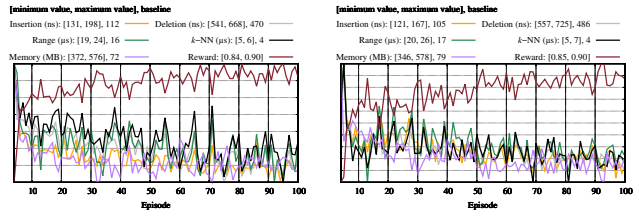


(a) x: $nCell^{space}_{lat}$ y: $nCell^{space}_{lon}$ (b) x: $nCell^{space}_{lat}$ y: $nCell^{chunk}_{lat}$

(c) x: $nCell^{space}_{lat}$ y: $nCell^{chunk}_{lon}$ (d) x: $nCell^{space}_{lat}$ y: $MOPC$

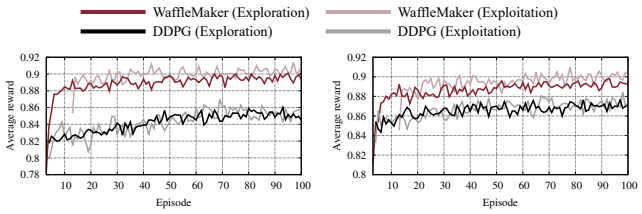
Figure 15: Relationships between the Waffle knobs (LA)

Figure 15 shows relationships between the Waffle knobs. For each $nCell^{space}_{lat}$ value, Waffle processed 1 episode of queries with the knob values in the y-axis, and for each query type, we recorded the value in the y-axis with the shortest processing time. The sample correlation coefficient [31] was calculated, and a regression line was estimated from the method of least squares. The default knob values were set to $\{1000, 1000, 10, 10, 10\}$ for knobs that do not appear on a graph. A clear pattern from the experiments was anti-correlation between $nCell^{space}_{lat}$ and $nCell^{space}_{lon}$ for the insertion queries, which means that the excessively small or large number of cells in a space was not preferred. However, formulation of the pattern was still difficult because the results may vary from various factors including object distribution and user queries. The experiment supports the necessity of an automatic configuration tuning system that does not require prior knowledge of the relationships.



(a) LA (b) NewYork

Figure 16: Validity of the reward definition in WaffleMaker



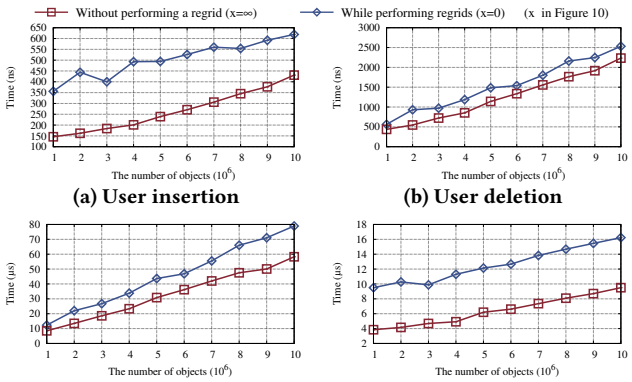
(a) LA (b) NewYork

Figure 17: Comparison with DDPG

7.2.2 Experiments for WaffleMaker and Regrid. We first introduce the experiments for WaffleMaker in Figures 16-17. Before each experiment, we generated queries for continued and different 100 episodes. When processing $x = 505,000$ user queries after the previous regrid (x in Figure 10), Waffle started another regrid. During the 100 episodes, approximately 2000 regrids were performed. At the beginning of each episode, DDPG initialized an Ornstein-Uhlenbeck process. For the initial $|batch|$ regrids, each method randomly determined knob settings without considering exploration and exploitation. During the initial regrids, WaffleMaker also recorded the minimum and maximum values for each reward element and utilized them for normalizing each element based on min-max normalization. DDPG also utilized the same minimum and maximum values as those of WaffleMaker. WaffleMaker started exploitation after the $|batch| + |recent|$ regrids. WaffleMaker and DDPG determined each knob value from the following ranges: $nCell^{space}_{lat,lon} : [100, 2000]$, $MOPC : [2, 20]$, and $nCell^{chunk}_{lat,lon} : [2, 20]$.

Figure 16 shows the validity of the reward definition. For each episode, we recorded average values of query processing times, memory usage, and rewards from knob settings determined by exploration. Memory usage was calculated using $|chunks| \times mChunk$ in Section 6.1. We also recorded the baselines for each query type and memory usage to measure the performance achievement of WaffleMaker. Specifically, we selected 10,000 random knob settings, executed one episode of queries for each knob setting, obtained query processing times and memory usage, and picked the best results as the baselines. Note that the best results did not come from a single knob setting, and obtaining the baselines took approximately 100 times longer than training WaffleMaker for the 100 episodes.

Initially, WaffleMaker obtained low rewards, which means that Waffle processed the queries slowly and required more memory. Based on the rewards, WaffleMaker updated the model and determined knob settings, and the query processing times and memory usage both decreased, which shows the validity of the reward definition. In addition, the memory usage was a lot larger than the baseline because reducing query processing time was the primary target ($w_{time} = 0.9, w_{memory} = 0.1$).



(a) User insertion (b) User deletion

(c) User range (d) User k-NN

Figure 18: Regrid overhead (LA)

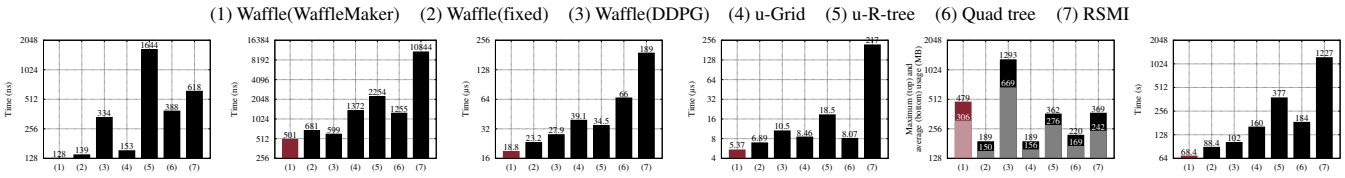
Table 1: Additional information about Figure 18 (LA)

Objects (10^6)	1	2	3	4	5	6	7	8	9	10
A single regrid time (seconds)	1.6	3.3	5.1	7.1	9.1	10.8	12.6	13.6	15.8	17.8
Concurrent user queries processed during a single regrid (10^5)	7.5	13.1	19.7	25.5	30.4	34.9	38.0	39.2	44.1	46.7
Total time to wait for locks for user queries during a single regrid (ms)	57	109	162	208	283	311	349	392	457	492

Figure 17 shows the rewards from exploration and exploitation for WaffleMaker and DDPG. (1) DDPG performed gradient ascent on the surface of an action-value function (i.e., critic) to update an actor. Because the critic kept updated, gradient ascent on the fluctuating surface tended to be unstable. WaffleMaker just observed the surface with enough candidate actions instead of performing gradient ascent, which led to higher rewards. (2) It was difficult to perfectly represent the surface of the critic. Given a set of experiences, a critic could learn a reasonable surface that represents high and low rewards. However, some parts of the surface had reverse slopes to what DDPG expected because the critic just tried to minimize the mean squared error for the experiences. Gradient ascent on the mis-represented surface resulted in lower rewards. In contrast, WaffleMaker carefully trusted and explored the surface based on weighted sampling for the knob settings with high expected rewards, which led to higher rewards. (3) Adding a noise to the current action did not explore action space efficiently, and DDPG was more likely to be stuck in a worse local minimum than WaffleMaker, whose exploration did not depend on the current policy. WaffleMaker discovered knob settings leading to high rewards that DDPG failed to find, learned a different action-value function from that of DDPG, and led to the higher rewards from exploitation. (4) In WaffleMaker, the rewards from exploitation were usually higher than those from exploration, which supported the validity of the novel exploitation method. (5) The average times for exploration/exploitation were 0.1008/0.022 s in WaffleMaker and 0.0021/0.0018 s in DDPG, respectively.

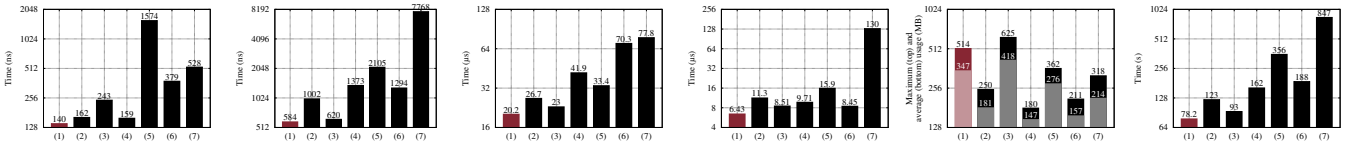
We also conducted the experiments to check the overhead of a regrid according to the number of objects ($|base|$). We set $|extra| = 0$ in the experiments. Figure 18 shows the average query execution times, and Table 1 shows additional information. We fixed the knob setting to $\{1000, 1000, 10, 10, 10\}$, which means that the knob setting might not be optimal for each case.

As shown in Figure 18, the performance of Waffle during a regrid was degraded because of additional cache misses from running the multiple threads. For a range/k-NN query during a regrid, Waffle



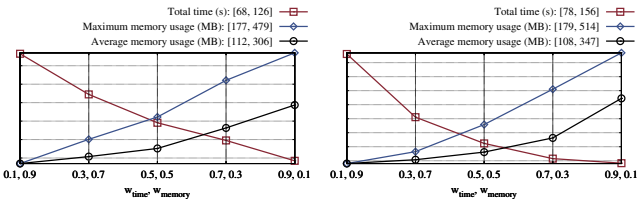
(a) Insertion (b) Deletion (c) Range (d) k-NN (e) Memory usage (f) Total time

Figure 19: Comparison with existing methods (LA)



(a) Insertion (b) Deletion (c) Range (d) k-NN (e) Memory usage (f) Total time

Figure 20: Comparison with existing methods (NewYork)



(a) LA (b) NewYork

Figure 21: Results for Waffle(WaffleMaker) according to w_{time} and w_{memory}.

processed two queries: one for the original index and another for the new index. However, the regrids finished within reasonable times based on the efficient insertion and deletion query processing without blocking the user queries, as shown in Table 1.

7.2.3 Experiments for Comparison with Existing Methods. Figures 19-20 show comparisons for Waffle(WaffleMaker), Waffle(fixed), Waffle(DDPG), u-Grid, u-R-tree, Quad tree, and RSMI. We generated 5 episodes of queries different from the 100 episodes in Section 7.2.2 for each dataset, where the four types of queries were mixed. For each method, we measured the average query execution time for each query type, maximum/average memory usage during runtime, and the total execution time on a log scale.

The experimental setting for each method was as follows. **(1) Waffle(WaffleMaker)**: WaffleMaker determined knob settings and when to perform a regrid as explained in Figure 11 (after convergence). We used the WaffleMaker models trained in Section 7.2.2. Some user queries were executed in parallel with a regrid as explained in Section 5 and also included in the results. **(2) Waffle(fixed)**: Waffle fixed a knob setting to the initial one used in Waffle(WaffleMaker) and did not perform regrids. **(3) Waffle(DDPG)**: DDPG determined knob settings instead of WaffleMaker. When a regrid started, DDPG obtained a candidate knob setting for the current state through exploitation. If the candidate was different from the current knob setting, Waffle performed the regrid. We used the DDPG models trained in Section 7.2.2. **(4) u-Grid**: u-Grid defined the index based on the initial knob setting used in Waffle(WaffleMaker). **(5) u-R-tree**: The minimum and maximum number of objects/children were set to 16 and 32, respectively. **(6) Quad tree**: The maximum number of objects in a leaf node was set to 64. If all the children of a non-leaf node were leaf nodes, and the total number of objects in the children was below 32, then the children were merged to the parent. **(7) RSMI**: Whenever RSMI reached the points where Waffle(WaffleMaker) performed the regrids, we interrupted query processing and rebuilt RSMI models. The building times including the initial build were excluded from the query processing times.

Waffle(WaffleMaker) provided the best performance for moving objects. The results supported the efficiency of a Waffle index defined from the concept of cells and chunks. In addition, the comparisons between Waffle(WaffleMaker) and Waffle(fixed) showed the benefit of performing regrids with appropriate knob settings automatically determined by WaffleMaker. Even if Waffle(fixed) utilized the decent knob settings from WaffleMaker, and performing regrids caused overhead to query processing as shown in Figure 18, Waffle(WaffleMaker) performed better than Waffle(fixed) because a fixed knob setting could not efficiently handle a variety of distribution of moving objects. However, Waffle(WaffleMaker) required more memory than Waffle(fixed). The fixed knob setting in Waffle(fixed) used less memory at several object distributions, where Waffle(WaffleMaker) changed knob settings focusing on query processing time. In addition, during a regrid, Waffle(WaffleMaker) kept two different indexes at the same time, which caused additional memory overhead.

Waffle(DDPG) performed worse than Waffle(WaffleMaker). The knob settings determined by DDPG were less efficient than those from WaffleMaker, as shown in Figure 17. Furthermore, when to perform regrids was important. Waffle(DDPG) performed 95/96 regrids for LA/NewYork while Waffle(WaffleMaker) performed 11/20 regrids. The query processing times of Waffle(DDPG) were even worse than Waffle(fixed) in some cases, which means that replacing continuous action space with discrete action space in WaffleMaker led to more timely regrids.

Although u-Grid also showed good performance, its performance was worse than that of Waffle(WaffleMaker). Even if a secondary index of u-Grid might reduce the computational cost by directly accessing a target bucket, additional cache misses occurred from updating the hash data structure for the secondary index. If there were a lot of buckets for the same cell coordinate, the overhead from the hash data structure might be compensated. However, Waffle tried to avoid the situation in which a lot of cells and chunks were created at the same coordinate by setting an appropriate

configuration from WaffleMaker. Even without performing regrids, Waffle(fixed) performed better than u-Grid in most cases, which showed the efficiency of a Waffle index.

u-R-tree accessed the target node directly with a secondary index without traversing the tree. However, because of the overhead from node splitting and merging, the performance of insertion and deletion queries were worse than that of u-Grid, which means that a grid-based index was more appropriate for moving objects than a tree-based index.

Quad tree performed worse than Waffle, which means that a uniform grid defined from the cell and chunk structure with appropriate configuration was more efficient than a multiple resolution grid for moving objects.

The observations for RSMI were as follows. (1) In RSMI, during query processing, figuring out target blocks required passing through RSMI models, which were on GPU in the experiments. In Waffle, figuring out target cells and chunks required few arithmetic operations, as shown in Section 2. Waffle utilized GPU not for calculating target cells and chunks during query runtime but for obtaining a new knob setting and updating the model. (2) Despite frequent rebuilding, RSMI performed worse than Waffle(WaffleMaker) for moving objects because the primary target of RSMI was non-moving objects, even if RSMI supported insertion and deletion queries. The total building times for LA/NewYork were 3.4/4.5 hours, respectively.

Figure 21 shows the results for Waffle(WaffleMaker) according to w_{time} and w_{memory} introduced in Section 6.1. WaffleMaker was trained for each pair of w_{time} and w_{memory} on the 100 episodes of queries used in Figures 16-17 and tested on the 5 episodes of queries used in Figures 19-20. If query processing time is more important than memory usage, w_{time} can be set higher than w_{memory}. If memory usage is more critical than query processing time, w_{memory} can be set higher than w_{time}. This direct and intuitive control for the trade-off between query processing time and memory usage was not available in the compared methods.

8 RELATED WORK

In-memory Spatial Index for Moving Objects. In-memory indexes for moving objects have been researched from various perspectives [7, 13, 30, 34–36, 43]. Darius et al. [35] proposed in-memory grid and R-tree indexes for moving objects. MOVIES [7] and TwinGrid [34] maintain an index for scan queries and another buffer/index for update queries. MOVIES and TwinGrid have a query staleness problem that some previous update query results may not be reflected in the following scan query results. PGrid [36] leverages the parallelism from a multi-core processor. PASTIS [30] considers the temporal dimension by maintaining the previous N days of data and indexes. D-Grid [43] considers both location and velocity information. SwapQt [13] operates in a cloud environment while reducing query staleness. The previous approaches, however, did not state how to automatically determine the index configuration, despite the configuration being significantly critical to their performance.

Learned Spatial Indexes. The indexing of spatial data based on machine learning has also emerged [12, 21, 29, 42]. ZM Index [42] and RSMI [29] are based on a recursive model index [15]. LISA [21] targets to reduce storage consumption and disk I/O cost. RLR-Tree [12] learns how to choose a subtree for an insertion query and split an overflowing node for an R-tree based on reinforcement learning to reduce disk I/O. SPRIG [45] learns spatial data distribution based on spatial interpolation function. In addition to the spatial indexes, the learning of a multidimensional index has been proposed in the studies [6, 27]. Waffle is a novel grid indexing system of which main target is an update query for moving objects, with automatic configuration tuning system, WaffleMaker.

Automatic Database Configuration. The automatic tuning of knob configuration in database systems has been studied based on machine learning. iTuned [8] utilizes Gaussian process regression to estimate performance from knob configuration. OtterTune [2] proposes a Gaussian process regression model with workload characterization and knob identification. The post-study on OtterTune [3] extends OtterTune to support not only Gaussian process regression but also deep neural networks and DDPG-based tuning methods. CDBTune [44] first applies DDPG to automatic database configuration based on a Markov decision process. QTune [19] is also based on DDPG considering query-level tuning. iBTune [39] and ResTune [47] configure resource-related knobs while guaranteeing the service level agreement for cloud databases. The study [40] utilizes natural language processing to obtain useful information from documents for configuration tuning. CGPTuner [4] considers knobs from databases and external layers including an operating system to improve the performance of databases based on contextual gaussian process bandit optimization. WaffleMaker is an online configuration tuning system that determines not only a knob setting but also times to change a configuration, without pre-training. In addition, the previous works [5, 17, 22, 28, 32] deal with an index selection problem based on machine learning.

9 CONCLUSION

We proposed Waffle, an in-memory grid indexing system for moving objects. A Waffle grid index is based on the cell and chunk definition optimized for the main memory. A knob configuration significantly affects the performance of Waffle, and an appropriate configuration depends on various factors. We proposed a regrid, a grid redefinition mechanism that does not block user queries based on a concurrency control scheme. To automatically determine an appropriate knob setting for a regrid, we introduced WaffleMaker, an online configuration tuning system based on novel reinforcement learning. The future works are as follows. Waffle can be extended to support continuous range or k-NN query processing optimized for Waffle considering the overhead. Distributed processing is also an area of future study.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This work was supported by (1) the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT (MSIT) (NRF-2020R1A2C2013286), (2) MSIT under the ICT Creative Consilience program (IITP-2022-2020-0-01819) supervised by the IITP (Institute for Information communications Technology Planning Evaluation), and (3) Basic Science Research Program through NRF funded by the Ministry of Education (NRF-2021R1A6A1A13044830).

REFERENCES

[1] Alekh Agarwal, Daniel J. Hsu, Satyen Kale, John Langford, Lihong Li, and Robert E. Schapire. 2014. Taming the Monster: A Fast and Simple Algorithm for Contextual Bandits. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014 (JMLR Workshop and Conference Proceedings)*, Vol. 32. JMLR.org, 1638–1646. http://proceedings.mlr.press/v32/agarwalb14.html

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1009–1024. https://doi.org/10.1145/3035918.3064029

[3] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *PVLDB* 14, 7 (2021), 1241–1253. http://www.vldb.org/pvldb/vol14/p1241-aken.pdf

[4] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: a Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations Under Varying Workload Conditions. *PVLDB* 14, 8 (2021), 1401–1413. https://doi.org/10.14778/3457390.3457404

[5] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1241–1258. https://doi.org/10.1145/3299869.3324957

[6] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *PVLDB* 14, 2 (2020), 74–86. https://doi.org/10.14778/3425879.3425880

[7] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. 2009. Indexing Moving Objects Using Short-Lived Throwaway Indexes. In *Advances in Spatial and Temporal Databases, 11th International Symposium, SSTD 2009, Aalborg, Denmark, July 8-10, 2009, Proceedings (Lecture Notes in Computer Science)*, Vol. 5644. Springer, 189–207. https://doi.org/10.1007/978-3-642-02982-0_14

[8] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *PVLDB* 2, 1 (2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[9] Ahmed Eldawy and Mohamed F. Mokbel. 2019. Roads and streets around the world each represented as individual line segments. https://doi.org/10.6086/N1H99379#mbr=9qwesvg4,9qxq6f81 Retrieved from UCR-STAR https://star.cs.ucr.edu/?OSM2015/road-network.

[10] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica* 4 (1974), 1–9. https://doi.org/10.1007/BF00288933

[11] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org/

[12] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2021. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *CoRR* abs/2103.04541 (2021). arXiv:2103.04541 https://arxiv.org/abs/2103.04541

[13] Hiba Jadallah and Zaher Al Aghbari. 2020. SwapQt: Cloud-based in-memory indexing of dynamic spatial data. *Future Generation Computer Systems* 106 (2020), 360–373. https://doi.org/10.1016/j.future.2020.01.009

[14] Sampath Kannan, Jamie Morgenstern, Aaron Roth, Bo Waggoner, and Zhiwei Steven Wu. 2018. A Smoothed Analysis of the Greedy Algorithm for the Linear Contextual Bandit Problem. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 2231–2241. https://proceedings.neurips.cc/paper/2018/hash/2cfd4560539f887a5e420412b370b361-Abstract.html

[15] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 489–504. https://doi.org/10.1145/3183713.3196909

[16] Akshay Krishnamurthy, John Langford, Aleksandrs Slivkins, and Chicheng Zhang. 2020. Contextual Bandits with Continuous Actions: Smoothing, Zooming, and Adapting. *The Journal of Machine Learning Research* 21 (2020), 137:1–137:45. http://jmlr.org/papers/v21/19-650.html

[17] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*. ACM, 2105–2108. https://doi.org/10.1145/3340531.3412106

[18] John Langford and Tong Zhang. 2007. The Epoch-Greedy Algorithm for Multi-armed Bandits with Side Information. In *Advances in Neural Information Processing Systems 20, Proceedings of the Twenty-First Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 3-6, 2007*. Curran Associates, Inc., 817–824. https://proceedings.neurips.cc/paper/2007/hash/4b04a686b0ad13dce35fa99fa4161c65-Abstract.html

[19] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *PVLDB* 12, 12 (2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[20] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. 2010. A Contextual-bandit Approach to Personalized News Article Recommendation. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*. ACM, 661–670. https://doi.org/10.1145/1772690.1772758

[21] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2119–2133. https://doi.org/10.1145/3318464.3389703

[22] Gabriel Paludo Licks, Júlia Mara Colleoni Couto, Priscilla de Fátima Miehe, Renata De Paris, Duncan Dubugras A. Ruiz, and Felipe Meneguzzi. 2020. SmartIX: A Database Indexing Agent Based on Reinforcement Learning. *Applied Intelligence* 50, 8 (2020), 2575–2588. https://doi.org/10.1007/s10489-020-01674-8

[23] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous Control with Deep Reinforcement Learning. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. http://arxiv.org/abs/1509.02971

[24] Tyler Lu, Dávid Pál, and Martin Pal. 2010. Contextual Multi-Armed Bandits. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010 (JMLR Proceedings)*, Vol. 9. JMLR.org, 485–492. http://proceedings.mlr.press/v9/lu10a.html

[25] Maryam Majzoubi, Chicheng Zhang, Rajan Chari, Akshay Krishnamurthy, John Langford, and Aleksandrs Slivkins. 2020. Efficient Contextual Bandits with Continuous Actions. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/033cc385728c51d97360020ed57776f0-Abstract.html

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[27] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 985–1000. https://doi.org/10.1145/3318464.3380579

[28] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving Index Tuning under Ad-hoc, Analytical Workloads with Safety Guarantees. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 600–611. https://doi.org/10.1109/ICDE51399.2021.00058

[29] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *PVLDB* 13, 11 (2020), 2341–2354. http://www.vldb.org/pvldb/vol13/p2341-qi.pdf

[30] Suprio Ray, Rolando Blanco, and Anil K. Goel. 2014. Supporting Location-Based Services in a Main-Memory Database. In *IEEE 15th International Conference on Mobile Data Management, MDM 2014, Brisbane, Australia, July 14-18, 2014 - Volume 1*. IEEE Computer Society, 3–12. https://doi.org/10.1109/MDM.2014.7

[31] Sheldon M Ross. 2020. *Introduction to Probability and Statistics for Engineers and Scientists*. Academic Press.

[32] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. DRLindex: Deep Reinforcement Learning Index Advisor for a Cluster Database. In *IDEAS 2020: 24th International Database Engineering & Applications Symposium, Seoul, Republic of Korea, August 12-14, 2020*. ACM, 11:1–11:8. https://dl.acm.org/doi/10.1145/3410566.3410603

[33] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. http://arxiv.org/abs/1511.05952

[34] Darius Sidlauskas, Kenneth A. Ross, Christian S. Jensen, and Simonas Saltenis. 2011. Thread-Level Parallel Indexing of Update Intensive Moving-Object Workloads. In *Advances in Spatial and Temporal Databases - 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24-26, 2011, Proceedings (Lecture Notes in Computer Science)*, Vol. 6849. Springer, 186–204. https://doi.org/10.1007/978-3-642-22922-0_12

[35] Darius Sidlauskas, Simonas Saltenis, Christian W. Christiansen, Jan M. Johansen, and Donatas Saulys. 2009. Trees or Grids?: Indexing Moving Objects in Main Memory. In *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*. ACM, 236–245. https://doi.org/10.1145/1653771.1653805

[36] Darius Sidlauskas, Simonas Saltenis, and Christian S. Jensen. 2012. Parallel Main-memory Indexing for Moving-object Query and Update Workloads. In

Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012. ACM, 37–48. https://doi.org/10.1145/2213836.2213842

[37] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016), 484–489. https://doi.org/10.1038/nature16961

[38] Aleksandrs Slivkins. 2011. Contextual Bandits with Similarity Information. In *COLT 2011 - The 24th Annual Conference on Learning Theory, June 9-11, 2011, Budapest, Hungary (JMLR Proceedings)*, Vol. 19. JMLR.org, 679–702. http://proceedings.mlr.press/v19/slivkins11a/slivkins11a.pdf

[39] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *PVLDB* 12, 10 (2019), 1221–1234. https://doi.org/10.14778/3339490.3339503

[40] Immanuel Trummer. 2021. The Case for NLP-Enhanced Database Tuning: Towards Tuning Tools that "Read the Manual". *PVLDB* 14, 7 (2021), 1159–1165. https://doi.org/10.14778/3450980.3450984

[41] George E Uhlenbeck and Leonard S Ornstein. 1930. On the Theory of the Brownian Motion. *Physical review* 36, 5 (1930), 823.

[42] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *20th IEEE International Conference on Mobile Data Management, MDM 2019, Hong Kong, SAR, China, June 10-13, 2019*. IEEE, 569–574. https://doi.org/10.1109/MDM.2019.00121

[43] Xiaofeng Xu, Li Xiong, and Vaidy S. Sunderam. 2016. D-Grid: An In-Memory Dual Space Grid Index for Moving Object Databases. In *IEEE 17th International Conference on Mobile Data Management, MDM 2016, Porto, Portugal, June 13-16, 2016*. IEEE Computer Society, 252–261. https://doi.org/10.1109/MDM.2016.46

[44] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 415–432. https://doi.org/10.1145/3299869.3300085

[45] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2021. SPRIG: A Learned Spatial Index for Range and kNN Queries. In *Proceedings of the 17th International Symposium on Spatial and Temporal Databases, SSTD 2021, Virtual Event, USA, August 23-25, 2021*. ACM, 96–105. https://doi.org/10.1145/3469830.3470892

[46] Wei Zhang, Jianzhong Li, and Haiwei Pan. 2006. Processing Continuous k-nearest Neighbor Queries in Location-dependent Application. *International Journal of Computer Science and Network Security* 6, 3 (2006), 1–9.

[47] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2102–2114. https://doi.org/10.1145/3448016.3457291