USN | | | | | | | | | | |

Internal Assessment Test 2 – DEC 2024

Sub:	INFORMATION RETRIEVAL					Sub Code:	BAI515B	Branch:	AIML
Date:	13 / 12/2024	Duration:	90 mins	Max Marks:	50	Sem / Sec:	V / A,B,C		OBE

Answer Any of 5 Questions

		MARKS	CO	RBT
1	Explain Aho-Corasick Algorithm for Pattern Searching. Build a Trie (or Keyword Tree) of all words. Input: text = "ahishers" arr[] = {"he", "she", "hers", "his"}	[10]	CO4	L3
2 (a)	Explain index compression with an example.	[05]	CO5	L2
(b)	Given a string s and string t, Apply Boyer-Moore algorithm to remove all occurrences of a string t in a string s Input: s = “abababababa”, t = “aba”	[05]	CO4	L3
3	Given a text txt[0..n-1] and a pattern pat[0..m-1] where n is the length of the text and m is the length of the pattern, Make use of a function pattern search(char pat[], char txt[]) that write all occurrences of pat[] in txt[]. Input: txt[] = “THIS IS A TEST TEXT” pat[] = “TEST”.	[10]	CO4	L3
4	Explain Multidimensional indexing with Suitable Example	[10]	CO4	L2
5.	Briefly explain about 1.search engine optimization. 2.A vector space model for XML retrieval	[10]	CO5	L2
6	How to build a Suffix Tree for a given text? “banana\0” where ‘\0’ is the string termination character.How to search a pattern in the built suffix tree	[10]	CO4	L3

CI CCI HOD-AIML

USN | | | | | | | | | | |

Internal Assessment Test 2 – DEC 2024

Sub:	INFORMATION RETRIEVAL					Sub Code:	BAI515B	Branch:	AIML
Date:	13 / 12/2024	Duration:	90 mins	Max Marks:	50	Sem / Sec:	V / A,B,C		OBE

Answer Any of 5 Questions

		MARKS	CO	RBT
1	Explain Aho-Corasick Algorithm for Pattern Searching. Build a Trie (or Keyword Tree) of all words. Input: text = "ahishers" arr[] = {"he", "she", "hers", "his"}	[10]	CO4	L3
2 (a)	Explain index compression with an example.	[05]	CO5	L2
(b)	Given a string s and string t, Apply Boyer-Moore algorithm to remove all occurrences of a string t in a string s Input: s = “abababababa”, t = “aba”	[05]	CO4	L3
3	Given a text txt[0..n-1] and a pattern pat[0..m-1] where n is the length of the text and m is the length of the pattern, Make use of a function pattern search(char pat[], char txt[]) that write all occurrences of pat[] in txt[]. Input: txt[] = “THIS IS A TEST TEXT” pat[] = “TEST”.	[10]	CO4	L3
4	Explain Multidimensional indexing with Suitable Example	[10]	CO4	L2
5	Briefly explain about 1.search engine optimization. 2.A vector space model for XML retrieval	[10]	CO5	L2
6	How to build a Suffix Tree for a given text? “banana\0” where ‘\0’ is the string termination character.How to search a pattern in the built suffix tree	[10]	CO4	L3

CI CCI HOD-AIML

USN | | | | | | | | | | |

Internal Assessment Test 2 – DEC 2024

Sub:	INFORMATION RETRIEVAL					Sub Code:	BAI515B	Branch:	AIML
Date:	13 / 12/2024	Duration:	90 mins	Max Marks:	50	Sem / Sec:	V / A,B,C		OBE

	Answer Any of 5 Questions	MARKS	CO	RBT
1	Explain Aho-Corasick Algorithm for Pattern Searching. Build a Trie (or Keyword Tree) of all words. Input: text = "ahishers" arr[] = {"he", "she", "hers", "his"} Definition-4 Drawing- 4 Explanation-2	[10]	CO4	L3
2 (a)	Explain index compression with an example. Definition-4 Explanation-1	[05]	CO5	L2
(b)	Given a string s and string t, Apply Boyer-Moore algorithm to remove all occurrences of a string t in a string s Input: s = “ababaababa”, t = “aba” Definition-4 Explanation-1	[05]	CO4	L3
3	Given a text txt[0..n-1] and a pattern pat[0..m-1] where n is the length of the text and m is the length of the pattern, Make use of a function pattern search(char pat[], char txt[]) that write all occurrences of pat[] in txt[]. Input: txt[] = “THIS IS A TEST TEXT” pat[] = “TEST”. Definition-4 Drawing- 4 Explanation-2	[10]	CO4	L3
4	Explain Multidimensional indexing with Suitable Example Definition-4 Drawing- 4 Explanation-2	[10]	CO4	L2
5.	Briefly explain about 1.search engine optimization. 2.A vector space model for XML retrieval Definition-4 Drawing- 4 Explanation-2	[10]	CO5	L2
6	How to build a Suffix Tree for a given text? “banana\0” where ‘\0’ is the string termination character.How to search a pattern in the built suffix tree Definition-4 Drawing- 4 Explanation-2	[10]	CO4	L3

CI CCI HOD-AIML

USN | | | | | | | | | | |

Internal Assessment Test 2 – Dec 2024

Sub:	INFORMATION RETRIEVAL					Sub Code:	BAI515B	Branch:	AIML	
Date:	13 / 12/2024	Duration:	90 mins	Max Marks:	50	Sem / Sec:	V / A, B, C		OBE	
Answer Any of 5 Questions								MARKS	CO	RBT

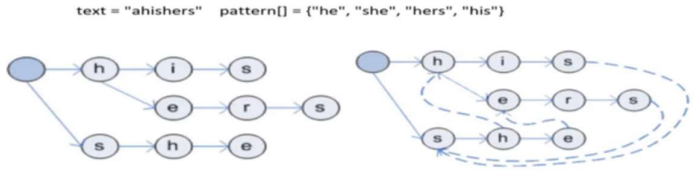
1.		[10]	CO1	L2

Explain Aho-Corasick Algorithm for Pattern Searching.

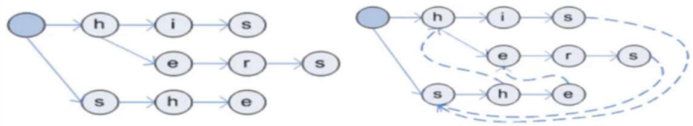
Build a Trie (or Keyword Tree) of all words.

Input text: “ahishers” arr = {“he”, she, “hers”, “his”}

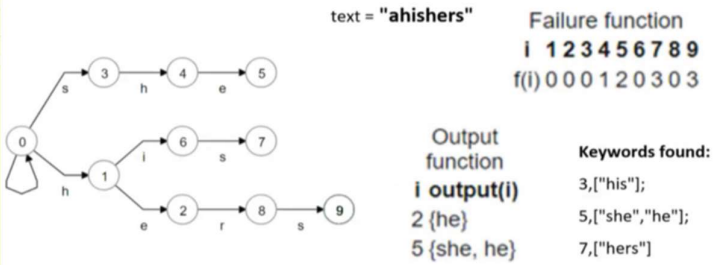
-Aho-Corasick algorithm is an **extension of the KMP algorithm** and is **specifically designed to efficiently search for multiple patterns in a text**.
-It is capable of handling a set of patterns **simulatenously**.
-Aho-Corasick Algorithm employs a **Trie-like data structure** to represent the set of patterns and employs a more **general set of failure transitions**.
-When a mismatch occurs during the pattern matching process, instead of restarting from the beginning,
Aho-Corasick utilizes the **failure transitions** to quickly jump to the **longest proper suffix** of the **currently matched prefix**.



Building of keyword tree (1 - after first step, 2 - tree with fail function)



Building of keyword tree (1 - after first step, 2 - tree with fail function)



2 (a)	Explain Index Compression with an Example. **Index Compression** Index compression is a technique used in information retrieval systems to reduce the size of an index while maintaining its efficiency for searching. Large-scale search engines and databases generate massive indexes to map words to documents or locations in text, and compressing these indexes is crucial to save storage space and improve query performance. Compression techniques are applied to both **inverted indexes** (used for storing postings lists) and **forward indexes**. The aim of index compression is to: 1. **Minimize storage requirements**. 2. **Reduce disk I/O latency** for faster access. 3. **Improve search performance** by reducing the index size. **Index Compression Techniques** **1. Vocabulary Compression** The vocabulary in an inverted index contains unique terms (words) that appear in the text corpus. Since natural language has redundancy, compressing vocabulary terms reduces space. **Techniques include**: • **Truncated Coding**: Use shorter codes for common words. • **Dictionary Encoding**: Store the vocabulary in sorted order and use relative storage for similar words. **2. Postings List Compression** The postings list stores document IDs (and positions) where a term appears. These lists can be very large and need compression. **Techniques include**: • **Gap Encoding**: Instead of storing absolute document IDs, store differences (gaps) between consecutive document IDs. • **Variable-Length Encoding**: Encode gaps using fewer bits for smaller numbers. • **Bit-Level Encoding**: Compress each integer to use minimal bits (e.g., Elias Gamma Encoding, Golomb Coding). **Example:** Consider the following documents: • **Doc 1**: *"The cat sat on the mat."* • **Doc 2**: *"The cat ate a rat."* **Step 1: Build the Inverted Index**	[05]	CO2	L2

The inverted index maps terms to the list of document IDs where they appear:

Term	Posting List
cat	1, 2
sat	1
on	1
the	1, 2
mat	1
ate	2
a	2
rat	2

Step 2: Apply Gap Encoding

Instead of storing absolute document IDs, we store **gaps** (differences) between consecutive IDs:

Term	Original Posting List	Gap Encoded List
cat	1, 2	1, 1
the	1, 2	1, 1
sat	1	1
on	1	1
mat	1	1
ate	2	2
a	2	2
rat	2	2

Explanation:

- For **cat**: Original list = [1, 2]. Gap Encoded List = [1, 1] (2 - 1 = 1).
- For **the**: Original list = [1, 2]. Gap Encoded List = [1, 1] (2 - 1 = 1).

Step 3: Variable-Length Encoding

To further compress the gap-encoded list, we use a variable-length encoding method such as **Elias Gamma Encoding**:

- Elias Gamma Encoding represents a number `n` as `1^(log2(n))0` followed by the binary representation of `n` without the leading 1.

For the gap-encoded list:

Term	Gap Encoded List	Elias Gamma Encoding
cat	1, 1	1, 1
the	1, 1	1, 1
sat	1	1
on	1	1
mat	1	1
ate	2	010
a	2	010
rat	2	010

Here:

- **1** → encoded as `1` (smallest number).
- **2** → encoded as `010`.

Step 4: Compress Vocabulary

The vocabulary can be further compressed using **front-coding**:

- Sort the terms alphabetically.
- Store the first word completely. For subsequent words, only the prefix difference is stored.

For example:

Sorted Terms	Stored Compression
ate	ate
a	1 a
cat	cat
mat	1 mat
on	1 on
rat	1 rat
sat	1 sat
the	the

Here, instead of repeating full words, **prefix differences** are stored after the first word.

2 (b)	**Boyer-Moore Algorithm to Remove All Occurrences of a Pattern** The **Boyer-Moore Algorithm** is an efficient string-matching algorithm that uses two heuristics: 1. **Bad Character Heuristic** 2. **Good Suffix Heuristic** It preprocesses the pattern to speed up the searching phase and skips unnecessary comparisons by aligning the pattern smartly against the text. **Problem Statement** Given: • Text t = "abababab" • Pattern s = "aba" We need to **remove all occurrences** of the pattern from the text using the Boyer-Moore algorithm. **Key Steps** 1. **Preprocessing**: Build the bad character table for the pattern. 2. **Search Phase**: Use the bad character heuristic to efficiently search for the pattern in the text and find all matches. 3. **Post Processing**: Remove all matches (occurrences of the pattern) from the text. **Step 1: Preprocessing - Bad Character Heuristic** The **bad character heuristic** shifts the pattern when a mismatch occurs. A table is constructed to store the last occurrence of every character in the pattern. For the given pattern s = "aba": • Build the **bad character table** as follows: **Character Last Occurrence (Index)** a 2 b 1 • If a mismatch occurs at a character, we check the bad character table to determine how far to shift the pattern. **Step 2: Search Phase - Pattern Search in the Text**	[05]	CO2	L2

We slide the pattern s over the text t and compare characters from right to left. If there is a mismatch, we use the **bad character heuristic** to shift the pattern.

Initialize:

- n = 8 → Length of text t = "abababab"
- m = 3 → Length of pattern s = "aba"

Procedure:

1. Align the pattern at the start of the text. Compare characters from right to left.
2. If a mismatch occurs, use the bad character heuristic to determine the shift.
3. If the pattern matches completely, record the position and continue searching further down the text.
4. Repeat until the entire text is processed.

Example Execution

Text: t = "abababab"
Pattern: s = "aba"

Initial Alignment:

Text: a b a b a b a b

Pattern: a b a

1. **Compare** from right to left:
    - t[2] = a matches s[2] = a
    - t[1] = b matches s[1] = b
    - t[0] = a matches s[0] = a

Match found at index **0**.

- Record the match position.
- Shift the pattern to continue searching.

Shift Pattern to Index 2:

Text: a b a b a b a b

Pattern: a b a

2. **Compare** from right to left:
    - t[4] = a matches s[2] = a
    - t[3] = b matches s[1] = b

- t[2] = a matches s[0] = a

Match found at index **2**.

- Record the match position.
- Shift the pattern further.

Shift Pattern to Index 4:

Text: a b a b a b a b

Pattern: a b a

3. **Compare** from right to left:
   - t[6] = a matches s[2] = a
   - t[5] = b matches s[1] = b
   - t[4] = a matches s[0] = a

Match found at index **4**.

- Record the match position.
- Shift the pattern further.

Shift Pattern to Index 6:

Text: a b a b a b

Pattern: a b a

4. **Compare** from right to left:
   - t[6] = a matches s[2] = a
   - t[7] = b → **Mismatch occurs**.

Use the bad character heuristic to shift the pattern. Since b is at index 1 in the bad character table, shift the pattern to align with the next possible position.

Final Matches

The pattern s = "aba" is found at positions **0, 2, and 4** in the text t.

Step 3: Remove All Occurrences

After identifying all positions where the pattern occurs (0, 2, and 4), remove the pattern aba from these positions.

	Original Text: abababab **After Removal**: • Remove aba starting at index 0 → Remaining text: bababab • Remove aba starting at index 2 → Remaining text: bab **Final Result**: Resultant Text: "b"			

3	Given a text txt[0..n-1] and a pattern pat[0..m-1] where n is the length of the text and m is the length of the pattern, write a function search(char pat[],char txt[]) that prints all occurrences of pat[] in txt[]. Input: txt[] = “THIS IS A TEST TEXT” pat[] = “TEST” **BadChar** Table:- T 3 E 1 S 2 ***Start aligning pat with txt**:- -Compare pat[3] with the corresponding text character. -On mismatch, shift the pattern based on the Bad Character Heuristic M = 4 N = 19 "THIS IS A TEST TEXT" "TEST" Mismatch on 'S' [Move (M-1)-2 = 1 Space] "THIS IS A TEST TEXT" "TEST" Mismatch on 'I' (Move (M) = 4 Spaces) "THIS IS A TEST TEXT" "TEST" Mismatch on ' ' (Move (M) = 4 Spaces) "THIS IS A **TEST** TEXT" "**TEST**" Matches, move one space over "THIS IS A TEST TEXT" "TEST" Mismatch on ' ' (Move (M) = 4 Spaces) "THIS IS A TEST TEXT" "TES**T**" Mismatch on 'X ' (Move (M) = 4 Spaces) END Pattern pat found once in the given string txt. **me set]**	[10]	CO2	L3

4	Briefly explain about:	[05]	CO3	L2
(a)	**Search engine optimization** SEO stands for search engine optimization. Let's break that down in the context of your website. • **Search:** What people do when they want to find an answer to a question or a product or service that meets their needs. • **Search engine:** A site (like Google or Bing) where a person can perform said search. • **Search engine optimization:** What you do to get said search engine to connect said search with your site. **A formal definition of SEO:** Search engine optimization is a set of technical and content practices aimed at aligning a website page with a search engine's ranking algorithm so it can be easily found, crawled, indexed, and surfaced in the SERP for relevant queries. **A simpler definition of SEO:** SEO is about making improvements to your website's structure and content so its pages can be discovered by people searching for what you have to offer, through search engines. **The simplest definition of SEO:** SEO is what you do to rank higher on Google and get more traffic to your site. Yes, Google is just one search engine of many. There's Bing. Directory search engines. Even Instagram is a search engine. But capturing 92% of the market share, the terms "Google" and "search engine" are synonymous for the intents and purposes of this post. **Benefits & importance of SEO** People are searching for any manner of things both loosely and directly related to your business. These are all opportunities to connect with these people, answer their questions, solve their problems, and become a trusted resource for them. • **More website traffic:** When your site is optimized for search engines, it gets more traffic which equates to increased brand awareness, as well as… • **More customers:** To get your site optimized, it has to target keywords—the terms your ideal customers/visitors are searching—meaning you'll get more relevant traffic. • **Better reputation:** Ranking higher on Google builds instant credibility for your business. If Google trusts you, then people trust you. • **Higher ROI:** You put money into your website, and into the marketing campaigns that lead back to your website pages. A top-performing site improves the fruits of those campaigns, making your investment worth it. **How does SEO work?**			

	So how does Google determine which pages to surface in the search engine results page (SERP) for any given query? How does this translate into traffic to your website? Let's take a look at how SEO works. • Google's search crawlers constantly scan the web, gathering, categorizing, and storing the billions of web pages out there in its index. When you search for something and Google pulls up results, it's pulling from its index, not the web itself. • Google uses a complex formula (called an algorithm) to order results based on a number of criteria (ranking factors—which we'll get into next) including the quality of the content, its relevance to the search query, the website (domain) it belongs to, and more. • How people interact with results then further indicates to Google the needs that each page is (or isn't) satisfying, which also gets factored into the algorithm.			

4 (b)	Vector space model for XML retrieval	[05]	CO2	L3

VECTOR SPACE MODEL

Representing XML Documents - Decompose text nodes into individual word nodes ,Use lexicalized subtrees as vector space dimensions , Captures both content and structure .

Querying and Retrieval - Represent queries as vectors in the same space .Use SIMNOMERGE, a modified cosine similarity function .SIMNOMERGE considers context resemblance between query and document paths.

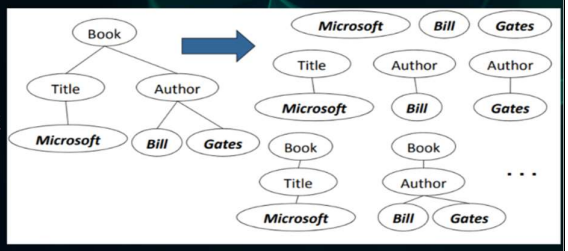
Relaxed Matching (SIMMERGE) - SIMMERGE relaxes matching conditions further collects term statistics from relevant contexts.

Main idea: lexicalized subtrees

Aim: to have each dimension of the vector space encode a word together with its position within the XML tree.
How: Map XML documents to lexicalized subtrees.

❶ Take each text node (leaf) and break it into multiple nodes, one for each word. E.g. split Bill Gates into Bill and Gates

❷ Define the dimensions of the vector space to be lexicalized subtrees of documents – subtrees that contain at least one vocabulary term



Document similarity measure

The final score for a document is computed as a variant of the cosine measure, which we call SIMNOMERGE. SIMNOMERGE(q, d) =

$$\sum_{c_k \in B} \sum_{c_l \in B} \text{CR}(c_k, c_l) \sum_{t \in V} \text{weight}(q, t, c_k) \frac{\text{weight}(d, t, c_l)}{\sqrt{\sum_{c \in B, t \in V} \text{weight}^2(d, t, c)}}$$

- V is the vocabulary of non-structural terms
- B is the set of all XML contexts
- weight (q, t, c), weight(d, t, c) are the weights of term t in XML context c in query q and document d, resp.

SIMNOMERGE(q, d) is not a true cosine measure since its value can be larger than 1.0.

5.	How to build a Suffix Tree for a given text? "banana\0" where \0 is the string termination character. How to search a pattern in the built Suffix Tree?	[10]	CO3	L2

Suffix Tree Structure

- A suffix tree is a **trie data structure built over all the suffixes of the text.**
- The pointers to the suffixes are stored at the **leaf nodes**
- This trie is compacted into a **Patricia tree (compressing unary paths).**

Searching

Many basic patterns such as words, prefixes, and phrases can be searched by a **simple trie search.**

Space Inefficiency:

Each trie node consumes 12 to 24 bytes, depending on the implementation.

Even with indexing only word beginnings, it **incurs a space overhead of 120% to 240% over the text size.**

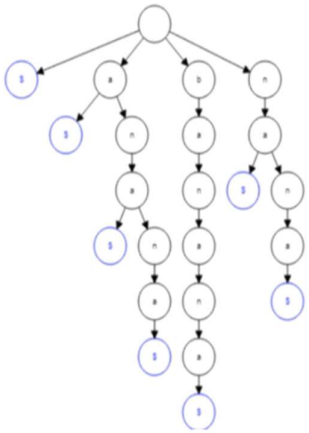
In **Information Retrieval (IR)**, text compression techniques are essential for reducing storage requirements and improving retrieval efficiency. Text compression can be divided into **two main categories**: **lossless** and **lossy** compression. Lossless compression maintains the exact original text, while lossy compression allows some information to be discarded to achieve higher compression. Below are the primary types of text compression techniques used in IR:

- Document Data is

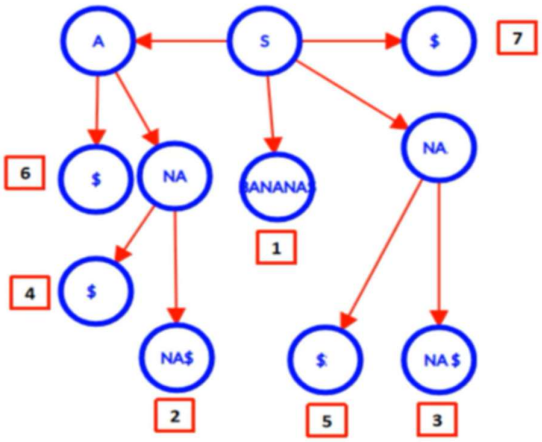
1	2	3	4	5	6	7
b	a	n	a	n	a	$

Suffix Tree For above data with the suffix words are as follows

Suffixes are **$, a$, na$, ana$, nana$, anana$, banana$,**



Suffix Trees



Text

1	2	3	4	5	6	7
b	a	n	a	n	a	$

Suffix Arrays

1	2	3	4	5	6	7
6	2	4	1	3	5	7
a	a	a	b	n	n	$
$	n	n	a	a	a	
	a	a	n	n	$	
	n	$	a	a		
	a		n	$		
	$		a			
			$			

6 A	Explain Multidimensional Indexing with Suitable Example. ❖ MULTIDIMENSIONAL INDEX • Multidimensional indexing is used to efficiently handle and query data with multiple attributes or dimensions, such as spatial data (e.g., latitude and longitude), temporal data (e.g., time series), or even high-dimensional feature vectors (e.g., machine learning). • The goal is to organize the data in a way that minimizes the search space when performing range queries, nearest-neighbor queries, or other complex operations. **KEY CONCEPTS IN MULTIDIMENSIONAL INDEXING** **Multidimensional Data:** • Data points have multiple attributes or dimensions (e.g., spatial coordinates like latitude and longitude, time, or other measurable attributes). • Ex: `(x, y)` for 2D spatial data or `(x, y, z, t)` for 3D space and time. **Objective:** • Enable efficient queries such as range queries (find all points within a region) or nearest-neighbor searches. **Indexing Structures:** • Data is partitioned or hierarchically structured to allow fast access to relevant regions of the data. STEPS IN MULTIDIMENSIONAL INDEXING **Data Preprocessing:** • Normalize and map multidimensional data into a suitable structure. • Ex: For spatial data, convert geographical coordinates into a Cartesian coordinate system. **Index Construction:** • Use a multidimensional data structure to organize data points: • Divide the space into partitions (e.g., grid cells or hierarchical regions). • Store data points in the corresponding partitions. **Query Execution:** • Traverse the index to find the relevant partitions based on the query (e.g., range or nearest-neighbor search). **Result Refinement:** • Extract and return data points from the relevant partitions.	[10]	CO3	L3

COMMON MULTIDIMENSIONAL INDEXING STRUCTURES

R-Tree:

- Organizes spatial data hierarchically using bounding rectangles.
- Ex: Query for points within a bounding box.

KD-Tree:

- Splits the data space into regions using a binary tree.
- Ex: Query for the nearest neighbor to a point.

Quadtrees:

- Divides 2D space recursively into quadrants.
- Ex: Query for all points in a specific quadrant.

Grid Indexing:

- Divides space into uniform grids.
- Ex: Query for all points in a grid cell.