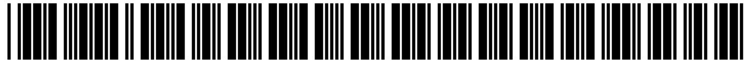

US 20060085592A1

(19) **United States**
(12) **Patent Application Publication**
(10) **Pub. No.: US 2006/0085592 A1**
(43) **Pub. Date: Apr. 20, 2006**

Ganguly et al.

(54) **METHOD FOR DISTINCT COUNT ESTIMATION OVER JOINS OF CONTINUOUS UPDATE STREAM**

(76) Inventors: **Sumit Ganguly**, Shaktinagar (IN); **Minos N. Garofalakis**, Morristown, NJ (US); **Amit Kumar**, New Delhi (IN); **Rajeev Rastogi**, New Providence, NJ (US)

Correspondence Address:
PATTERSON & SHERIDAN, LLP/ LUCENT TECHNOLOGIES, INC
595 SHREWSBURY AVENUE
SHREWSBURY, NJ 07702 (US)

(21) Appl. No.: **10/957,185**

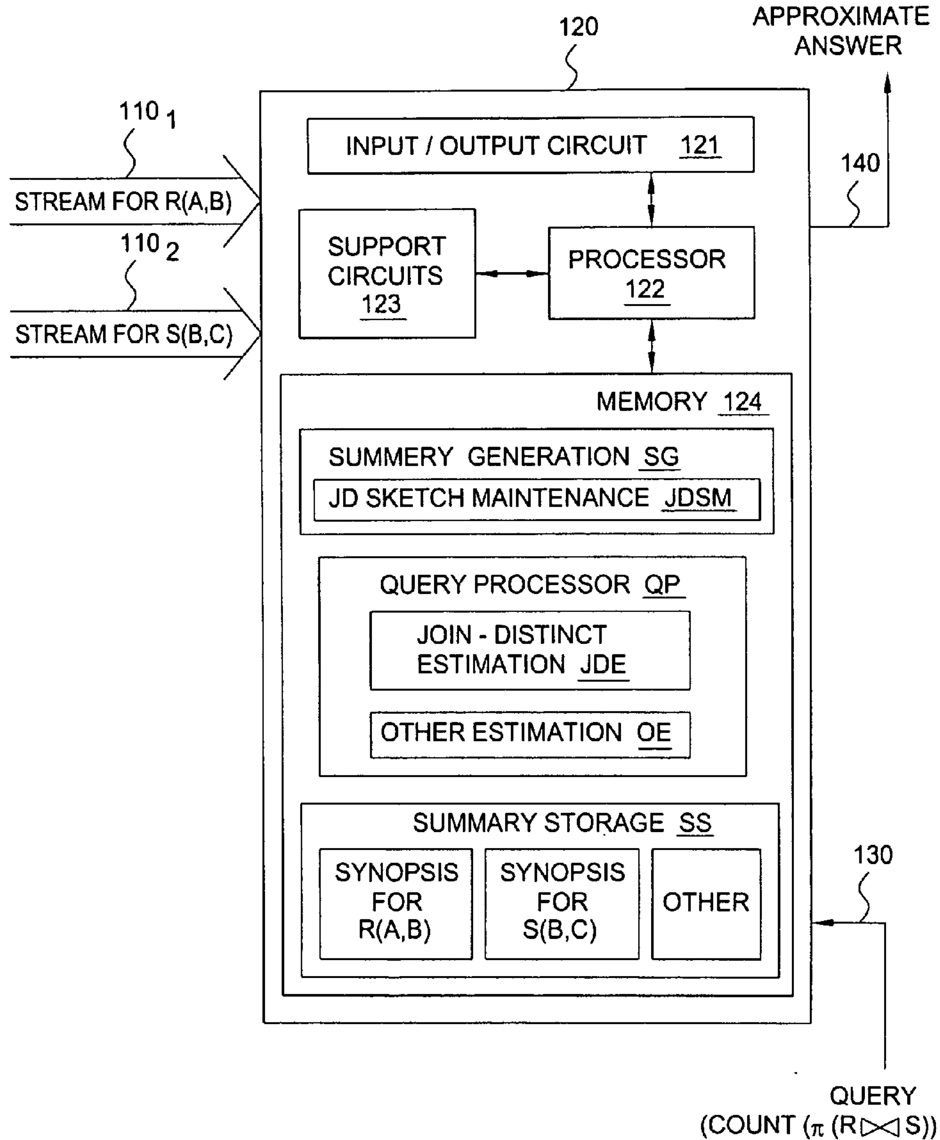
(22) Filed: **Sep. 30, 2004**

Publication Classification

(51) **Int. Cl.**
G06F 12/14 (2006.01)
(52) **U.S. Cl.** **711/114**

(57) **ABSTRACT**

The invention provides methods and systems for summarizing multiple continuous update streams using corresponding multiple (parallel) JD Sketch data structures such that, for example, an approximate answer to a query requiring a join operation followed by a duplicate elimination step may be rapidly provided.



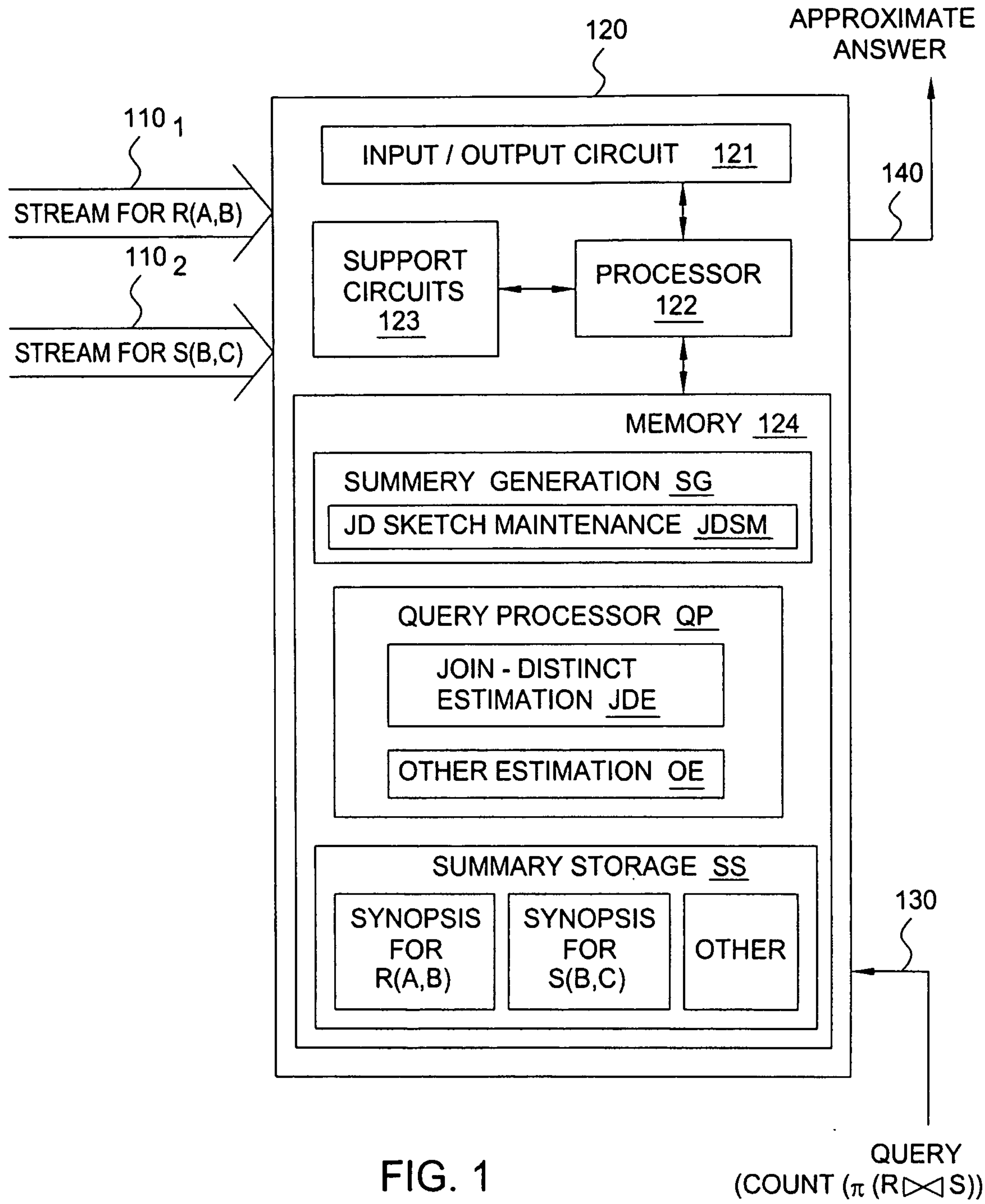
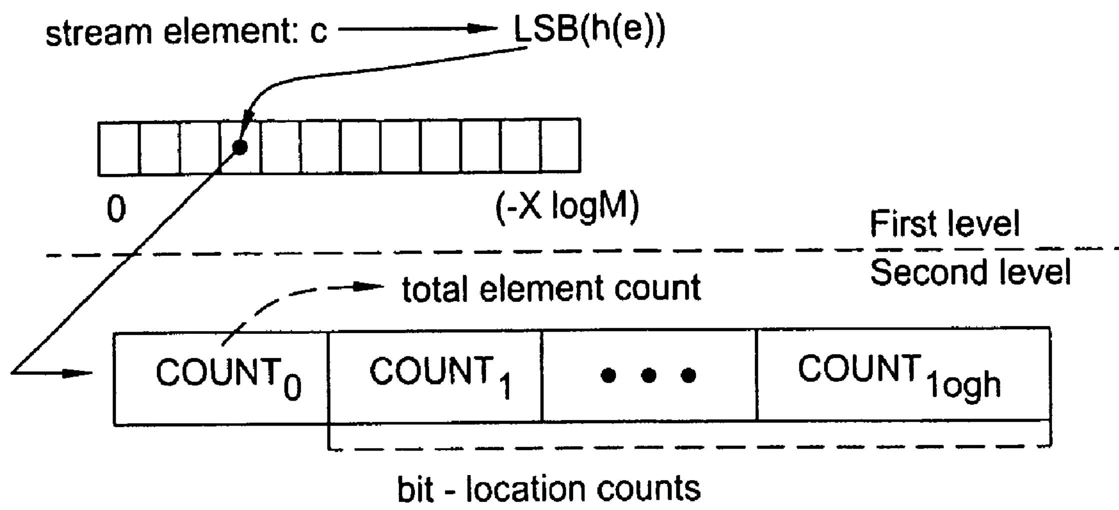
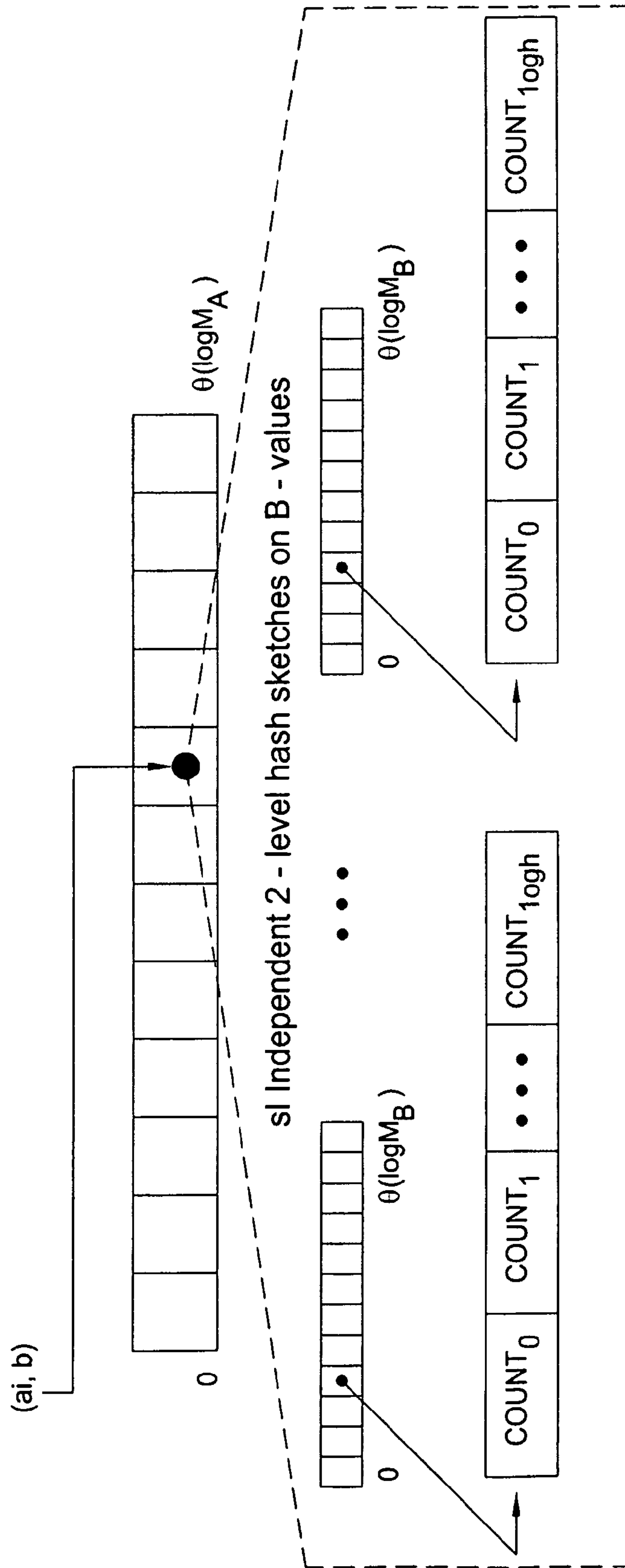


FIG. 1



The 2-level Hash Sketch Synopsis Structure.

FIG. 2



JD Sketch Synopsis for Stream R (A,B).

FIG. 3

procedure Compose($\mathcal{X}_{A,B}, \mathcal{X}_{C,B}$)
Input: Pair of parallel JD sketch synopses for update streams R(A,B), S(B,C).
Output: Bitmap-sketch $\mathcal{Y}_{A,C}$ on (A, C) value pairs in R⋈S
begin
1. $\mathcal{Y}_{A,C} := [0, \cdots, 0]$ // initialize
2. **for each** bucket k of $\mathcal{X}_{A,B}$ **do**
3. **for each** bucket $l \neq k$ of $\mathcal{X}_{C,B}$ **do**
4. **if** IntersectionEstimator($\mathcal{X}_{A,B}[k]$, $\mathcal{X}_{C,B}[l]$) ≥ 1 **then**
5. $\mathcal{Y}_{A,C}[\min\{k, l\}] := 1$
6. **return**($\mathcal{Y}_{A,C}$)
end

JD Sketch Composition Algorithm.

FIG. 4

procedure JDEstimator($(\mathcal{X}^i_{A,B}, \mathcal{X}^i_{C,B}), i = 1, \ldots, s_2, \epsilon$)
Input: s_2 independent pairs of parallel JD sketch synopses $(\mathcal{X}^i_{A,B}, \mathcal{X}^i_{C,B})$ for the update streams R(A,B), S(B,C).
Output: (ϵ, δ)-estimate for $|\pi_{A,C}(R(A,B) \bowtie S(B,C))|$.
begin
1. **for** $i := 1$ **to** s_2 **do**
2. $\mathcal{Y}^i_{A,C}$:= Compose($\mathcal{X}^i_{A,B}, \mathcal{X}^i_{C,B}$)
3. let B denote the highest bucket index in the $\mathcal{Y}_{A,C}$ synopses
4. **for** index := B **downto** 0 **do**
5. count := 0
6. **for** $i := 1$ **to** s_2 **do**
7. **if** $\mathcal{Y}^i_{A,C}$[index] $= 1$ **then** count := count +1
8. **endfor**
9. **if** $\left((1 - 2\epsilon)\frac{\epsilon}{8} \leq \frac{\text{count}}{s_2} \leq (1 + \epsilon)\epsilon \right)$ **then**
10. **return**(count/$(p_{\text{index}} \cdot s_2)$)
11. **endfor**
12. **return**(fail)
end

Join-Distinct Estimation Algorithm.

FIG. 5

METHOD FOR DISTINCT COUNT ESTIMATION OVER JOINS OF CONTINUOUS UPDATE STREAM

FIELD OF THE INVENTION

[0001] The invention relates generally to information processing systems and, more particularly, to database query processing over continuous streams of update operations.

BACKGROUND OF THE INVENTION

[0002] Query-processing algorithms for conventional Database Management Systems (DBMS) typically rely on several passes over a collection of static data sets in order to produce an accurate answer to a user query. However, there is growing interest in algorithms for processing and querying continuous data streams (i.e., data that is seen only once in a fixed order) with limited memory resources. These streams in general comprise update operations (insertions, deletions and the like).

[0003] Providing even approximate answers to queries over continuous data streams is a requirement for many application environments; examples include large IP network installations where performance data from different parts of the network needs to be continuously collected and analyzed. A large network processes data traffic and provides measurements of network performance, network routing decisions and the like. Other application domains giving rise to continuous and massive update streams include retail-chain transaction processing (e.g., purchase and sale records), ATM and credit-card operations, logging Web-server usage records, and the like.

[0004] For example, assume that each of two routers within the network provides a respective update stream indicative of packet related data, router behavior data and the like. It may be desirable for the data streams from each of the two routers to be correlated. Traditionally, such streams are correlated using a JOIN operation, which is used to determine, for example, how many of the tuples associated with routers R**1** and R**2** have the same destination IP address (or some other inquiry). In the case of this JOIN query, the two data sets (i.e., those associated with R**1** and R**2**) are joined and the size of the relevant joined set is determined (e.g., how many of the tuples have the same destination address).

[0005] The ability to estimate the number of distinct (sub)tuples in the result of a join operation correlating two data streams (i.e., the cardinality of a projection with duplicate elimination over a join) is an important goal. Unfortunately, existing query processing solutions are unable to provide sufficient responses to complex "Join-Distinct" estimation problems over data streams.

SUMMARY OF THE INVENTION

[0006] Various deficiencies in the prior art are addressed by a novel method and data structure for summarizing a continuous update stream. Where the data structure is used to summarize multiple continuous update streams, approximate answers to Join-Distinct queries and other queries may be rapidly provided. Improved accuracy in query response is achieved in one embodiment by summarizing multiple continuous data streams using corresponding multiple (parallel) JD Sketch data structures. One embodiment of the invention is directed to determining a "distinct" join aggregate. That is, the invention operates to perform a join operation, then apply a duplicate elimination step to count the number of distinct tuples produced by the join operation.

[0007] Specifically, a method according to one embodiment of the invention comprises maintaining a summary of a first continuous stream of tuples by hashing tuples received from the first continuous data stream according to at least one initial attribute; and for each bucket of the at least one initial attribute, generating a corresponding set of 2-level hash sketches according to at least one other attribute.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] The teachings of the present invention can be readily understood by considering the following detailed description in conjunction with the accompanying drawings, in which:

[0009] **FIG. 1** depicts an Update-Stream Processing Architecture according to an embodiment of the present invention;

[0010] **FIG. 2** depicts a data structure of a 2-level hash sketch synopsis useful in understanding the invention;

[0011] **FIG. 3** depicts a data structure of a JD Sketch update stream synopsis according to an embodiment of the invention;

[0012] **FIG. 4** depicts a pseudo code representation of a method according to an embodiment of the invention for composing parallel JD Sketch data structures; and

[0013] **FIG. 5** depicts a pseudo code representation of a Joint-Distinct estimation method according to an embodiment of the invention.

[0014] To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to the figures.

DETAILED DESCRIPTION OF THE INVENTION

[0015] The invention will be described within the context of a computer or communications network in which the provisioning and relationships of various routers and other data are of interest. It will be appreciated that the invention is broadly applicable to any system or application domain in which the efficient summarizing and, optionally, Join-Distinct query processing of multiple continuous update data streams is desired.

[0016] Briefly, presented herein is a space-efficient algorithmic solution to the general Join-Distinct cardinality estimation problem over continuous update streams. The proposed estimators are probabilistic in nature and rely on novel algorithms for building and combining a new class of hash-based synopses, termed "JD sketches", that are useful in summarizing update streams. Also presented are novel estimation algorithms that use the JD sketch synopses to provide low error, high-confidence Join-Distinct estimates using only small space and small processing time per update.

[0017] First, the JD sketch synopsis data structure is introduced and its maintenance over a continuous stream of updates (rendering a multi-set of data tuples) is described. Briefly, the JD synopses make use of 2-level hash sketch

structures while imposing an additional level of hashing that is adapted to effectively project and count on the attributes in the distinct-count clause. The JD sketch synopses never require rescanning or re-sampling of past stream items, regardless of the deletions in the stream: at any point in time, the synopsis structure is guaranteed to be identical to that obtained if the deleted items had never occurred in the stream. The JD sketch synopses are used to summarize multiple continuous data streams such that approximate answers to Join-Distinct queries and other queries may be rapidly provided.

[0018] Second, based on the JD sketch synopses, a novel probabilistic algorithm for estimating Join-Distinct cardinalities over update streams is provided. That is, the invention operates to perform a join operation, then apply a duplicate elimination step to count the number of distinct tuples produced by the join operation. A key element of the solution is a new technique for intelligently composing independently-built JD sketches (on different tuple streams) to obtain an estimate for the cardinality of their Join-Distinct result. The invention provides a novel approach to solve this difficult estimation problem in the context of a data-streaming or update-streaming model. Furthermore, even though the estimators are presented in a single-site setting, the invention may be extended to the more general distributed-stream model.

[0019] A method according to an embodiment of the invention operates to build small space summaries of each of the update streams of interest. That is, as the individual update records or tuples stream by, a small space summary of these records/tuples is built and maintained for each stream. The small space summary for each stream is supported by a respective plurality (e.g., Z) of a new data structure denoted by the inventors as a JD Sketch synopsis. In response to a user query, the invention operates to process the small space summaries (i.e., the Z parallel JD Sketch data structures) of the streams relevant to the user query according to a Join-Distinct operation to provide thereby very good approximate response(s) to the query.

[0020] Each of the Z JD Sketch data structures associated with any update stream represents a respective synopsis of that update stream up to a particular point in time. Each of the Z JD Sketch data structures (JDS_1-JDS_Z) is determined in a consistent manner between streams. That is, assuming two streams of interest, the first JD Sketch JDS_1 of a first stream STREAM**1** is constructed in the same manner as the first JD Sketch JDS_1 of a second stream STREAM**2**, the second JD Sketch JDS_2 of the first stream STREAM**1** is constructed in the same manner as the second JD Sketch JDS_2 of the second stream STREAM**2** and so on. Thus, each of the records or tuples received from an update stream is hashed and stored according to each of the Z JD Sketch synopsis. The Z JD Sketches associated with each update stream forms a small space summary of that update stream.

[0021] The JD Sketch data structure is not a standard hash table. It may be conceptualized as a hash table having buckets which additionally include second level hash structures adapted to capture information pertaining to the set of elements of a stream. That is, only a summary of the set of information mapping to each hash table bucket is actually kept in the bucket. This summary hash table bucket summary may comprise, illustratively, a set of 2-level hash sketch synopses. Thus, each hash table bucket stores a corresponding set of hash tables of those data elements that map or "collide" into the hash table bucket.

[0022] As previously noted, each of the corresponding 1-Z JD Sketch data structures maintained for each stream is constructed in the same manner. That is, for each of the update streams, their small space summaries are built using corresponding JD Sketch data structures. Specifically, the first level hash function and set of second level hash functions for a first JD Sketch data structure JDS**1** of a first stream STREAM**1** corresponds to the first level hash function and set of second level hash functions for a first JD Sketch data structure JDS**1** of a second stream STREAM**2** (and so on). Such corresponding JD Sketch pairs are herein referred to as parallel JD Sketch pairs.

[0023] In their most general form, data base data streams are actually update streams; that is, the stream is a sequence of updates to data items, comprising data-item deletions as well as insertions. Such continuous update streams arise naturally, for example, in the network installations of large Internet service providers, where detailed usage information (SNMP/RMON packet-flow data, active VPN circuits, etc.) from different parts of the underlying network needs to be continuously collected and analyzed for interesting trends.

[0024] The processing of such streams follows, in general, a distributed model where each stream (or, part of a stream) is observed and summarized by its respective party (e.g., the element-management system of an individual IP router) and the resulting synopses are then collected (e.g., periodically) at a central site, where queries over the entire collection of streams can be processed. This model is used, for example, in the Interprenet IP network monitoring product provided by Lucent Technologies, Inc., of Murray Hill, N.J.

[0025] There are several forms of queries that users or applications may wish to pose (online) over such continuous update streams; examples include join or multi-join aggregates, norm and quantile estimation, or histogram and wavelet computation. Estimating the number of distinct (sub-)tuples in the result of an equi-join operation correlating two update streams (i.e., the cardinality of a projection with duplicate elimination over a join) is one of the fundamental queries of interest for several data-analysis scenarios.

[0026] As an example, a network-management application monitoring active IP-sessions may wish to correlate the active sessions at routers R**1** and R**2** by posing a query such as: "What is the number of distinct (source, destination) IP-address pairs seen in packets routed through R**1** such that the source address is also seen in packets routed by R**2**?" Such a query would be used, for example, when trying to determine the load imposed on a core router R**1** by the set of customers connected to a specific router R**2** at the edge of the network. This query may be described as the number of distinct tuples in the output of the following project-join query, where $R_i(sour_i, dest_i)$ denotes the multi-set of source-destination address pairs observed in the packet stream through a router R_i, as follows:

$$\pi_{sour_1, dest_1}(R_1(sour_1, dest_1) \bowtie_{sour_1 = sour_2} R_2(sour_2, dest_2))$$

[0027] The ability to provide effective estimates for the cardinality of such "Join-Distinct" query expressions over the observed IP-session data streams in the underlying

network can be crucial in quickly detecting possible denial-of-service attacks, network routing or load-balancing problems, potential reliability concerns (catastrophic points-of-failure), and so on. Join-Distinct queries are also an integral part of query languages for relational database systems (e.g., the DISTINCT clause in the SQL standard). Thus, one-pass synopses for effectively estimating Join-Distinct cardinalities can be extremely useful, e.g., in the optimization of such queries over Terabyte relational databases.

[0028] **FIG. 1** depicts an Update-Stream Processing Architecture according to an embodiment of the present invention. The architecture of **FIG. 1** is presented as a general purpose computing element adapted to perform the various stream processing tasks described herein. It will be appreciated by those skilled in the art and informed by the present invention that the architecture of **FIG. 1** may be usefully employed within the context of a network management system at a Network Management Layer, an Element Management Layer or some other level. Moreover, while not discussed in detail herein, appropriate systems and apparatus for practicing the data structures, methodology and other aspects of the invention may be found in any system benefiting from the data processing and other techniques described herein.

[0029] Specifically, **FIG. 1** comprises an update-stream processing architecture **120** including a processor **122** as well as memory **124** for storing various control programs and other programs as well as data. The memory **124** may also store an operating system supporting the various programs.

[0030] The processor **122** cooperates with conventional support circuitry such as power supplies, clock circuits, cache memory and the like as well as circuits that assist in executing the software routine stored in the memory **124**. As such, it is contemplated that some of the steps discussed herein as software processes may be implemented within hardware, for example as circuitry that cooperates with the processor **122** to perform various steps. The processing architecture **120** also contains input/output (I/O) circuitry **121** which forms an interface between the various functional elements communicating with the architecture **120**.

[0031] The architecture **120** may be advantageously employed within the context of a network management system (NMS), an element management system (EMS) or any other network management system. Similarly, the invention has broad applicability to any system in which large amounts of data must be rapidly processed within the context of update streams.

[0032] The invention may be implemented as a computer program product wherein computer instructions, when processed by a computer, adapt the operation of the computer such that the methods, data structures and/or techniques of the present invention are invoked or otherwise provided. Instructions for invoking the inventive methods may be stored in fixed or removable media, transmitted via a data stream in a broadcast media, and/or stored within a working memory within a computing device operating according to the instructions.

[0033] In an embodiment of the invention depicted in **FIG. 1**, a first update-stream $\mathbf{110}_1$ is associated with a first multi-set of relational tuples R(A,B), while a second update-stream $\mathbf{110}_2$ is associated with a second multi-set of relational tuples denoted as S(B,C). The update-stream processing architecture **120** processes the two update streams $\mathbf{110}_1$ through $\mathbf{110}_2$ to produce corresponding summaries. The corresponding summaries are the processed in response to, illustratively, a user query **130** to provide therefrom an approximate answer **140**.

[0034] Memory **124** is depicted as including a summary generation SG algorithm is utilized to provide summary information pertaining to the received update streams, illustratively a JD sketch maintenance algorithm JDSM. The summary (sets of parallel JD sketch pairs) for each of the update streams $\mathbf{110}_1$, $\mathbf{110}_2$ is stored within summary storage SS within memory **124**. As depicted in **FIG. 1**, the summary storage SS includes a synopsis for stream R(A,B), a synopsis for stream S(B,C) and, optionally, a synopsis for other update streams. Thus, while the architecture of **FIG. 1** is depicted as processing two update streams, more update streams may be summarized and otherwise processed in accordance with the present invention.

[0035] Memory **124** is also depicted as including a query processor QP including a join-distinct estimation algorithm (JDE) as well as another estimation algorithm OE. The JDE algorithm also utilizes a JD Sketch composition algorithm. Thus, summaries in the form of JD Sketchs are generated/maintained by the summary generation SG algorithm. The summaries/JD Sketches are then composed and otherwise utilized by the JDE algorithm to respond to queries.

[0036] It will be appreciated by those skilled in the art and informed by the teachings of the present invention that while the memory **124** includes a plurality of data structures, algorithms and storage regions, there is no requirement within the context of the present invention that a single memory device as depicted be utilized within the context of the update-stream processing architecture. Specifically, any combination of internal, external and/or associated memory may be utilized to store the software instructions necessary to provide summary generation SG functions, query processor QP functions and summary storage SS functions. Thus, while the architecture depicts the memory as an integral portion of a relatively unified structure, the memory **124** may in fact be distributed internal or external to the update-stream processing architecture **120**.

[0037] The update-stream processing architecture of **FIG. 1** is adapted in one embodiment for Join-Distinct cardinality estimation. Each input stream renders a multi-set of relational tuples (R(A,B) or S(B,C)) as a continuous stream of updates. Note that, in general, even though A, B, and C are denoted as single attributes in the model described herein, they can in fact denote sets of attributes in the underlying relational schema. Furthermore, without loss of generality, it will assumed for convenience that each attribute $X \in \{A,B,C\}$ takes values from the integer domain $[M_X]=\{0, \ldots, M_X-1\}$. Each streaming update (illustratively, for input R(A,B)) is a pair of the form $<(a,b),\pm v>$, where $(a,b) \in [M_A] \times [M_B]$ denotes the specific tuple of R(A,B) whose frequency changes, and $\pm v$ is the net change in the frequency of (a,b) in R(A,B), i.e., "$\pm v$" to ("$-v$") denotes v insertions (resp., deletions) of tuple (a,b). It is assumed that all deletions in the update streams are legal; that is, an update $<(a,b),-v>$ can only be issued if the net frequency of (a,b) in R(A,B) is at least v. Also let N denote an upper bound on the total number

of data tuples (i.e., the net sum of tuple frequencies) in either R(A,B) or S(B,C). In contrast to conventional DBMS processing, the inventive stream processor is likely allowed to see the update tuples for each relational input only once and in the fixed order of arrival as they stream in from their respective source(s). Backtracking over an update stream and explicit access to past update tuples are impossible.

[0038] The invention finds particular applicability within the context of estimating the number of distinct(A,C) (sub-)tuples in the result of the data-stream join $R(A,B) \bowtie_B S(B,C)$. More specifically, in approximating the result of the query $Q=|\pi_{A,C}(R(A,B) \bowtie S(B,C))|$ or, using SQL:

```
Q =        SELECT COUNT DISTINCT (A, C)
           FROM R(A,B), S(B,C)
           WHERE R.B = S.B
```

[0039] The term $|X|$ is used to denote the set cardinality (i.e., number of distinct elements with positive net frequency) in the multi-set X. In general, the attribute sets A, B, and C in Q are not necessarily disjoint or non-empty. For example, the target attributes A and C may in fact contain the join attribute B, and either A or C can be empty (i.e., a one-sided projection). To simplify the discussion, the estimation algorithms assume that both A and C are non-empty and disjoint from B (i.e., $A,C \neq \phi$ and $A \cap B = B \cap C = \emptyset$). The invention is also applicable to other forms of Join-Distinct estimation. With respect to notation, the term A is used as a shorthand to denote the set of distinct A-values seen in R(A,B), and $|A|$ to denote the corresponding set cardinality (i.e., $|A|=|\pi_A(R(A,B))|$). (B,C and $|B|$,$|C|$ are used similarly, with B being the distinct B-values seen in either R(A,B) or S(B,C), i.e., the union $\pi_B(R(A,B)) \cup \pi_B(S(B,C))$).

[0040] **FIG. 2** depicts a data structure of a 2-level hash sketch update stream synopsis useful in understanding the invention. Specifically, the subject invention utilizes a data structure that includes a 2-level hash sketch stream synopsis. In one embodiment, the 2-level hash sketch stream synopsis is formed according to a modification of a technique developed by Flajolet and Martin (FM). The FM technique proposed the use of hash-based techniques for estimating the number of distinct elements (i.e., $|A|$) over an insert-only data stream A (i.e., a data stream without deletions or other non-insert updates). Briefly, assuming that the elements of A range over the data domain [M], the FM algorithm relies on a family of hash functions H that map incoming elements uniformly and independently over the collection of binary strings in [M]. It is then easy to see that, if $h \in H$ and (s) denotes the position of the least-significant 1 bit in the binary string s, then for any

$$i \in [M], (h(i)) \in \{0, \ldots, \log M - 1\} \text{ and } = \frac{1}{2^{l+1}}.$$

The basic hash synopsis maintained by an instance of the FM algorithm (i.e., a specific choice of hash function $h \in H$) is simply a bit-vector of size $\Theta(\log M)$. This bit-vector is initialized to all zeros and, for each incoming value i in the input (multi-)set A, the bit located at position (h(i)) is turned on. Of course, to boost accuracy and confidence, the FM algorithm employs averaging over several independent instances (i.e., r independent choices of the mapping hash-function $h \in H$ and corresponding synopses). The key idea behind the FM algorithm is that, by the properties of the hash functions in H, it is expected that a fraction of

$$\frac{1}{2^{l+1}}$$

of the distinct values in A to map to location l in each synopsis; thus, it is expected that $|A|/2$ values to map to bit **0**, $|A|/4$ to map to bit **1**, and so on. Therefore, the location of the leftmost zero (say A) in a bit-vector synopsis is a good indicator of $\log|A|$, or, $2^\lambda \approx |A|$.

[0041] A generalization of the basic FM bit-vector hash synopsis, termed a 2-level hash sketch, enables accurate, small-space cardinality estimation for arbitrary set expressions (e.g., including set difference, intersection, and union operators) defined over a collection of general update streams (ranging over the domain [M], without loss of generality). 2-level hash sketch synopses rely on a family of (first-level) hash functions H that uniformly randomize input values over the data domain [M]; then, for each domain partition created by first-level hashing, a small (logarithmic-size) count signature is maintained for the corresponding multi-set of stream elements.

[0042] More specifically, a 2-level hash sketch uses one randomly-chosen first-level hash function $h \in H$ that, as in the FM algorithm, is used in conjunction with LSB operator to map the domain elements in [M] onto a logarithmic range $\{\mathbf{0}, \ldots, \Theta(\log M)\}$ of first-level buckets with exponentially decreasing probabilities. Then, for the collection of elements mapping to a given first-level bucket, a count signature comprising an array of log M+1 element counters is maintained. This count-signature array consists of two parts: (a) one total element count, which tracks the net total number of elements that map onto the bucket; and, (b) log M bit-location counts, which track, for each l=1, . . . , log M, the net total number of elements e with $_l(e)=1$ that map onto the bucket (where, $_l(e)$ denotes the value of the 1th bit in the binary representation of $e \in [M]$). Conceptually, a 2-level hash sketch for a streaming multi-set A can be seen as a two-dimensional array S_A of size $\Theta(\log M) \times (\log M+1) = \Theta(\log^2 M)$, where each entry $S_A[k,l]$ is a data-element counter of size $O(\log N)$ corresponding to the lth count-signature location of the kth first-level hash bucket. Assume that, for a given bucket k, $S_A[k,0]$ is always the total element count, whereas the bit-location counts are located at $S_A[k,1]$, . . . , $S_A[k, \log M]$. The structure of this 2-level hash sketch synopses is pictorially depicted in **FIG. 2**.

[0043] The algorithm for maintaining a 2-level hash sketch synopsis S_A over a stream of updates to a multi-set A operates as follows. The sketch structure is first initialized to all zeros and, for each incoming update $\langle e, \pm v \rangle$, the element counters at the appropriate locations of the SA sketch are updated; that is, set $S_A[(h(e)),0] := S_A[(h(e)),0] \pm v$ to update the total element count in e 's bucket and, for each l=1, . . . , log M such that $_l(e)=1$, set $S_A[(h(e)),l] := S_A[(h(e)),l] \pm v$ to update the corresponding bit-location counts. Note here that the 2-level hash sketch synopses are essentially impervious

to delete operations; in other words, the sketch obtained at the end of an update stream is identical to a sketch that never sees the deleted items in the stream.

Join-Distinct Synopsis Data Structure

[0044] **FIG. 3** depicts a data structure of a JD Sketch update stream synopsis according to an embodiment of the invention. Each JD Sketch data structure is associated with one first level hash function and a set of second level hash functions. Multiple JD Sketch data structures are maintained in parallel. That is, each record/tuple is processed according to the requirements of each of the Z JD Sketch data structures. Each respective Z JD Sketch data structure represents a small-space summary of an update data stream.

[0045] The JD sketch synopsis data structure for update stream R(A,B) uses hashing on attribute(s) A (similar, in one embodiment, to the basic FM distinct-count estimator) and, for each hash bucket of A, a family of 2-level hash sketches is deployed as a concise synopsis of the B values corresponding to tuples mapped to this A-bucket. More specifically, a JD sketch synopsis $X_{A,B}$ for stream R(A,B) relies on a hash function h_A selected at random from an appropriate family of randomizing hash functions H_A that uniformly randomize values over the domain $[M_A]$ of A. As in the FM algorithm (and 2-level hash sketches), this hash function h_A is used in conjunction with the operator to map A-values onto a logarithmic number of hash buckets $\{0, \ldots, \Theta)(\log M_A)\}$ with exponentially-decreasing probabilities. Each such bucket $X_{A,B}[i]$ is an array of s_1 independent 2-level hash sketches built on the (multi-)set of B values for (A,B) tuples whose A component maps to bucket i. Let $X_{A,B}[i,j]$ $(1 \leqq j \leqq s_1)$ denote the ith 2-level hash sketch on B for the ith A bucket. One aspect of the JD sketch definition is that the B hash functions (h_B) used by the ith 2-level hash sketch in $X_{A,B}$ are, in one embodiment, identical across all A buckets. That is, $X_{A,B}[i_1,j]$ and $X_{A,B}[i_2,j]$ use the same (first-level) hash functions on B for any i_1,i_2 in $\{0, \ldots, \Theta(\log M_A)\}$.

[0046] As with 2-level hash sketches, conceptually, a JD sketch $X_{A,B}$ for the update stream R(A,B) can be seen as a four-dimensional array of total size $\Theta(\log M_A) \times s_1 \times \Theta(\log M_B) \times (\log M_B+1) = s_1 \cdot \Theta(\log M_A \log^2 M_B)$, where each entry $X_{A,B}[i,j,k,l]$ is a counter of size O(log N). The JD sketch structure is pictorially depicted in **FIG. 3**. It should be noted that a JD sketch $X_{A,B}$ can be thought of as a three-level hash sketch where the first level of hashing is on attribute(s) A, and a subsequent set of 2-level hash sketches on the B attribute(s).

[0047] The maintenance algorithm for a JD sketch synopsis built over the R(A,B) stream operates as follows. All counters in the data structure are initialized to zeros and, for each incoming update $<(a,b),\pm v>$ (where $(a,b) \in [M_A] \times [M_B]$), the a value is hashed using h_A ( ) to locate an appropriate A-bucket, and all the 2-level hash sketches in that bucket are then updated using the $<b,\pm v>$ tuple; that is, each of the s_1 2-level hash sketches $X_{A,B}[h_A(a),j]$ $(i=1, \ldots, s_1)$ is updated with $<b,\pm v>$ using the 2-level hash sketch maintenance algorithm described herein.

Join-Distinct Estimator

[0048] A Join-Distinct estimation algorithm according to an embodiment of the invention constructs several independent pairs of parallel JD sketch synopses $(X_{A,B}, X_{C,B})$ for the input update streams R(A,B) and S(B,C) (respectively). For the $X_{C,B}$ sketch, attribute C plays the same role as attribute A in $X_{A,B}$ (i.e., it is used to determine a first-level bucket in the JD sketch). Furthermore, both $X_{A,B}$ and $X_{C,B}$ use exactly the same hash functions for B in corresponding 2-level hash sketches for any A or C bucket; that is, the B-hash functions for $X_{A,B}[*,j]$ and $X_{C,B}[*,j]$ are identical for each $i=1, \ldots, s_1$ (here, "*" denotes any first-level bucket in either of the two JD sketches). Then, at estimation time, each such pair of parallel JD sketches is composed in a novel manner to build a synopsis for the number of distinct (A,C) pairs in the join result. This composition is novel and non-trivial, and relies on the use of new, composable families of hash functions (h_A( ) and h_C( )) for the first level of a JD sketch synopses. The basic JD sketch composition step will now be described in detail.

Composing a Parallel JD Sketch Pair.

[0049] Consider a pair of parallel JD sketch synopses $(X_{A,B}, X_{C,B})$ over R(A,B) and S(B,C). The goal of the JD sketch composition step is to combine information from $X_{A,B}$ and $X_{C,B}$ to produce a bitmap synopsis for the number of distinct (A,C) value pairs. This bitmap is built using only (A,C) pairs in the result of the join $R \bowtie S$. Thus, the composition step uses $X_{A,B}$ and $X_{C,B}$ to determine (with high probability) the (A,C)-value pairs that belong in the join result, and map such pairs to the cells of a bitmap $Y_{A,C}$ of logarithmic size (i.e., $O(\log M_A + \log M_C)$) with exponentially-decreasing probabilities.

[0050] **FIG. 4** depicts a pseudo code representation of a method for composing parallel JD Sketch data structures. The JD sketch composition algorithm of **FIG. 4**, termed Compose considers all possible combinations of fist level bucket indices $k \neq l$ from the two input parallel JD sketches (the $k \neq l$ restriction comes from a technical condition on the composition of the two first-level hash functions on A and C). For each such bucket pair $X_{A,B}[k]$ and $X_{C,B}[l]$, the composition algorithm employs the corresponding 2-level hash sketch synopses on B to estimate the size of the intersection for the sets of B values mapped to those first-level buckets (procedure IntersectionEstimator in Step **4**). If that size is estimated to be greater than zero (i.e., the two buckets share at least one common join-attribute value), then the bit at location $\min\{k,l\}$ of the output (A,C)-bitmap $Y_{A,C}$ is set to one (Step **5**). Since the JD sketches do not store the entire set of B values for each first-level bucket but, rather, a concise collection of independent 2-level hash sketch synopses, the decision of whether two first-level buckets join is necessarily approximate and probabilistic (based on the 2-level hash sketch intersection estimate).

[0051] The JD sketch composition algorithm implements a composite hash function $h_{A,C}$( ) over (A,C)-value pairs that combines the first-level hash functions h_A( ) and h_C( ) from sketches $X_{A,B}$ and $X_{C,B}$, respectively.

[0052] The composite hash function and its properties will now be examined in more detail. The ability to use the final (A,C) bitmap synopsis $Y_{A,C}$ output by algorithm Compose to estimate the number of distinct (A,C)-value pairs in $R(A,B) \bowtie (B,C)$ depends on designing a composite hash function $h_{A,C}$( ) (based on the individual functions h_A( ), h_C( )) that guarantees certain randomizing properties similar to those of the hash functions used in the 2-level hash sketch estimators. More specifically, the composite hash function $h_{A,C}$( ) preferably (a) allows mapping of (A,C)-value pairs

onto a logarithmic range with exponentially-decreasing probabilities, and (b) guarantees a certain level of independence between distinct tuples in the (A,C)-value pair domain. The key problem is that, since the tuples from R(A,B) and S(B,C) are seen in arbitrary order and are individually summarized in the $X_{A,B}$ and $X_{C,B}$ synopses, the composite hash-function construction can only use the hash values (h_A( )) and (h_C( )) maintained in the individual JD sketches. This limitation makes the problem non-trivial, since it reduces the efficacy of standard pseudo-random hash-function constructions, such as using finite-field polynomial arithmetic over $[M_A]\times[M_C]$.

[0053] The inventors have established the existence of composable hash-function pairs (h_A( ), h_C( )) and demonstrate that, for such functions, the composition procedure in algorithm Compose indeed guarantees the required properties for the resulting composite hash function (i.e., exponentially-decreasing mapping probabilities as well as pair wise independence for (A,C)-value pairs).

[0054] Specifically, the hash functions (h_A( ),h_C( )) used to build a parallel JD sketch pair ($X_{A,B}$,$X_{C,B}$) can be constructed so that the hash-function composition procedure in algorithm Compose: (1) guarantees that (A,C)-value pairs are mapped onto a logarithmic range $\Theta(\log \max\{M_A,M_C\})$ with exponentially-decreasing probabilities (in particular, the mapping probability for the ith bucket is $p_i=\Theta(4^{-(i+1)})$); and, (2) results in a composite hash function $h_{A,C}$( ) that guarantees pair wise independence in the domain of (A,C)-value pairs.

The Join-Distinct Estimator.

[0055] **FIG. 5** depicts a pseudo code representation of a Joint-Distinct estimation method according to an embodiment of the invention. Specifically, **FIG. 5** depicts the pseudo-code of an algorithm for producing an (ϵ,δ) probabilistic estimate for the Join-Distinct problem (termed JDEstimator). Briefly, the estimator employs an input collection of s_2 independent JD sketch synopsis pairs built in parallel over the input update streams R(A,B) and S(B,C). Each such parallel JD sketch pair ($X^i_{A,B}$,$X^i_{C,B}$) is first composed (using algorithm Compose) to produce a bitmap synopsis $Y^i_{A,C}$ over the (A,C)-value pairs in the join result (Steps **1**-**2**). Then, the resulting s_2 $Y^i_{A,C}$ bitmaps are examined level-by-level in parallel, searching for the highest bucket level ("index") at which the number of bitmaps that satisfy the condition: "$Y^i_{A,C}$[]=1", lies between

$$s_2\cdot(1-2\epsilon)\frac{\epsilon}{8} \text{ and } s_2\cdot(1+\epsilon)\epsilon \text{ (Steps 4-11).}$$

The final estimate returned is equal to the fraction of $Y^i_{A,C}$ synopses satisfying the condition at level "index" (i.e., "count/s_2") scaled by the mapping probability for that level $p=\Theta(4^{-(i+1)})$.

[0056] The inventors have established that for appropriate settings of the JD Sketch synopsis parameter s_1 and s_2, the join-distinct estimation procedure can guarantee low relative error with high probability. Specifically, let S_p denote a random sample of distinct (A,C)-value pairs drawn from the Cartesian product $[M_A]\times[M_C]$ of the underlying value domains, where each possible pair is selected for inclusion in the sample with probability p. Also, define two additional random variables U_p and T_p over the sample S_p of (A,C)-value pairs as follows:

[0057] Up=|{b: ∃(a,c)∈Sp such that [(a,b) ∈R(A,B) OR (b,c) ∈S(B,C)]}|,

[0058] Tp=|{b: ∃(a,c)∈Sp such that [(a,b) ∈R(A,B) AND (b,c) ∈S(B,C)]}|.

[0059] The following theorem summarizes the results of the analysis, using $M=\max\{M_A,M_B,M_C\}$ to simplify the statement of the worst-case space bounds:

[0060] Let m denote the result size of an input query on two update streams R(A,B) and S(B,C), and let

$$M=\max\{M_A,M_B,M_C\},\ e(p)=E\left[\frac{Up}{Tp}\middle|Tp\geq 1\right],\text{ where } p=\frac{\epsilon}{m}.$$

The algorithm JDEstimator returns an (ϵ,δ)-estimate (i.e., an estimate that is within an ϵ relative error of the correct answer with probability at least 1-δ) for m using JD Sketch synopses with a total storage requirement of $\Theta(s_1 s_2 \log^3 M \log N)$ and per-tuple update time of

$$\Theta(s_1 \log M),\text{ where: } s_1=\Theta\left(\frac{e(p)\log(1/\delta)}{\varepsilon}\right)\text{ and } s_2=\Theta\left(\frac{\log(1/\delta)}{\epsilon^3}\right).$$

Extensions

[0061] Handling Other Forms of Join-Distinct Count Queries. The discussion thus far has focused on the query $Q=|\pi_{A,C}(R(A,B)\bowtie S(B,C))|$, where $A,C\neq\phi$ and $A\cap B=B\cap C=\phi$). The Join-Distinct estimation techniques can be adapted to deal with other forms of Join-Distinct COUNT queries conforming to the general query pattern described herein.

[0062] Consider the case of a one-sided projection query $Q'=|\pi_{A,B}(R(A,B)\ltimes S(B,C))|$, where the estimate of the number of distinct R(A,B) tuples joining with at least one tuple from S(B,C) (i.e., the number of distinct tuples in a stream semi-join) is sought. The JDEstimator algorithm can readily handle the estimation of Q' by replacing attribute C with B in the JD sketch construction and estimation steps already described for Q. Thus, for Q', the JD sketch synopsis built on the S(B,C) side uses a first-level hash function on attribute B in addition to the per-bucket 2-level hash sketch collections (also built on B); then, when the JD sketch composition process is applied (at estimation time), the output is a set of bitmap synopses $Y_{A,B}$ on (A,B)-value pairs that is used to produce an estimate for Q'.

[0063] Similarly consider the case of a "full-projection" query $Q''=|\pi A,B,C(R(A,B)\bowtie S(B,C)|)$ that simply outputs the number of distinct (A,B, C) tuples in the join result. Handling Q" involves replacing A(C) by (A,B) (resp., (B,C)) in the JD sketch construction and JDEstimator algorithms for Q. The results of the analysis for the estimators can also be readily extended to cover these different forms of Join-Distinct estimation problems.

[0064] The Join-Distinct estimation algorithms described thus far have primarily utilized logarithmic or polylogarith-

mic space requirements. Such a restriction makes sense, for example, when joining on attributes with very large numbers of distinct values (e.g., (source, destination) IP-address pairs). When this is not the case, and using $\Theta(|B|)$ space is a viable option for estimating $Q=|\pi_{A,C}(R(A,B)\bowtie S(B,C))$, an alternative Join-Distinct estimation algorithm may be used. Briefly, the alternative algorithm again relies on the idea of using composable hash functions to compose a bit-vector sketch on (A,C) from hash sketches built individually on R(A,B) and S(B,C); however, the synopsis structure used is different from that of JDEstimator. More specifically, by making use of a $\Theta(|B|)$ bit-vector indicating the existence of a particular B value in an input stream; for each non-empty B-bucket, a collection of independent FM synopses (using counters instead of bits) is maintained that summarizes the collection of distinct A(C) values for tuples in R(A,B) (resp., S(B,C)) containing this specific B-value. (These FM synopses are built using composable hash functions $h_A(\ )$ and $h_C(\ )$, as discussed above. At estimation time, the A and C synopses for each B-value that appears in both R(A,B) and S(B,C) (note that, since by using $\Theta(|B|)$ space, this co-occurrence test is now exact) are composed to produce an (A,C)-bitmap sketch for that B-value. Then, all such (A,C)-bitmaps are unioned (by bitwise OR-ing) to give bit-vectors on (A,C) for the result of $R\bowtie S$, that can be directly used for estimating Q. This alternative Join-Distinct estimator can produce an (E,δ)-estimate for Q using

$$\Theta\left(|B|\log\left(\frac{1}{\delta}\right)\Big/\epsilon^3\right)$$

space, and can be extended to handle other forms of Join-Distinct queries (like Q' and Q" above).

[0065] The above-described invention provides a methodology for summarizing a continuous stream of tuple updates using the JD Sketch data structure. Multiple JD Sketch data structure may be used to summarize a tuple update stream, where each of the JD Sketch data structures is computed using a respective base attribute. Two (or more) tuple update streams may be summarized in parallel using respective multiple JD Sketch data structures, where pairs (i.e., one associated with each stream and having a common attribute) of JD Sketch data structures are maintained in parallel. The parallel JD Sketch data structures enable rapid approximations in response to join-distinct queries by performing a join operation and then applying a duplicate elimination step to count the number of distinct tuples produced by the join operation.

[0066] While the forgoing is directed to various embodiments of the present invention, other and further embodiments of the invention may be devised without departing from the basic scope thereof. As such, the appropriate scope of the invention is to be determined according to the claims, which follow.

1. A NOR flash memory device comprising:

a cell array including banks each composed of sectors, each sector being constructed of memory cells coupled to wordlines and bitlines;

a row selector to designate one of the wordlines in each bank in response to a row address;

a column selector to designate bitlines in a unit of a predetermined number in each bank in response to a column address;

a data input buffer to receive and hold program data bits in the unit of the predetermined number or less; and

a program driver to contemporaneously apply a program voltage to the designated bitlines in response to the program data bits held in the data input buffer.

2. The NOR flash memory device as set forth in claim 1, wherein the data input buffer is comprised of unit buffers each assigned to the banks.

3. The NOR flash memory device as set forth in claim 2, wherein each unit buffer receives the program data bits in the units of the predetermined number in parallel.

4. The NOR flash memory device as set forth in claim 1, wherein the program driver is supplied with a high voltage greater than a power source voltage to generate the program voltage.

5. The NOR flash memory device as set forth in claim 4, wherein the program driver supplies the program voltage from the high voltage or a ground voltage in response to the program data bits.

6. The NOR flash memory device as set forth in claim 2, wherein the program driver is comprised of unit drivers each assigned to the banks.

7. The NOR flash memory device as set forth in claim 1, wherein the predetermined number is 16.

8. The NOR flash memory device as set forth in claim 1, further comprising a fail detector to compare data bits of the cell array with the program data bits of the data input buffer.

9. The NOR flash memory device as set forth in claim 8, wherein the fail detector is shared by all of the banks.

10. The NOR flash memory device as set forth in claim 1, wherein the bitlines are constructed of local bitlines connected to the memory cells, and global bitlines connected to the local bitlines.

11. A system comprising:

a host to generate data bits in units of a first predetermined number in a second predetermined number of times or less; and

a NOR flash memory device operable in a program mode with the data bits supplied from the host, comprising:

a cell array including banks disposed in correspondence with the second predetermined number of times, each bank being composed of sectors, each sector being constructed of memory cells coupled to wordlines and bitlines;

a row selector to designate one of the wordlines in each bank in response to a row address;

a column selector to designate bitlines in units of the first predetermined number in each bank in response to a column address;

a data input buffer to receive and hold the data bits in the units of the first predetermined number in the second predetermined number of times or less; and

a program driver to contemporaneously apply a program voltage to the designated bitlines in response to the data bits held in the data input buffer.

12. The system as set forth in claim 11, wherein the program driver is supplied with a high voltage greater than a power source voltage from the host.

13. The system as set forth in claim 11, further comprising a fail detector to compare data bits of the cell array with the data bits of the data input buffer.

14. The system as set forth in claim 13, wherein the fail detector is shared by all of the banks.

15. The system as set forth in claim 11, wherein the first predetermined number is 16.

16. A method of programming a NOR flash memory device including a cell array with a number n banks each composed of sectors each of which is constructed of memory cells coupled to wordlines and bitlines, a row selector for designating one of the wordlines in each bank in response to a row address, and a column selector for designating units of i bitlines in each bank in response to a column address, the method comprising:

receiving a command to enable a program operation of i data bits at the same time;

receiving addresses to designate locations to store the data bits;

receiving and temporarily holding the i data bits an n number of times or less, corresponding to the designated bitlines; and

simultaneously applying a program voltage to the designated bitlines in response to the data bits temporarily held.

17. The method as set forth in claim 16, further comprising, prior to receiving the command segmenting entire program data into units of n banks and supplying the i data bits n times or less.

18. The method as set forth in claim 16, further comprising comparing data bits of the cell array with data bits of a data input buffer and detecting a fail of programming.

19. The method as set forth in claim 18, further comprising repeating applying the program voltape and comparing the data bits of the cell array until the data bits of the cell array are identical to the data bits temporarily held.

20. The method as set forth in claim 16, wherein the command to enable the program operation of the i data bits includes receiving the i data bits in parallel.

21. The method as set forth in claim 16, wherein simultaneously applying a program voltage includes applying the program voltage to the selected bitlines by receiving an external high voltage that is greater than a power source voltage.

22. The method as set forth in claim 21, wherein simultaneously applying a program voltage includes supplying the high voltage to the designated bitlines as the program voltage when the data bits are "0", and supplies a ground voltage to deselected bitlines when the data bits are "1".

23. The method as set forth in claim 16, wherein i is 16.

24. A NOR flash memory device comprising:

a cell array including banks each composed of sectors, each sector being constructed of memory cells coupled to wordlines and bitlines;

a row selector to designate one of the wordlines in each bank in response to a row address, the row selector including one row decoder for each one of the sectors;

a column selector to designate bitlines in a unit of a predetermined number in each bank in response to a column address, the column selector including one column decoder for each one of the sectors;

a data input buffer to receive and hold program data bits in the unit of the predetermined number or less; and

a program driver to contemporaneously apply a program voltage to the designated bitlines in response to the program data bits held in the data input buffer.

25. The NOR flash memory device as set forth in claim 24, wherein the row decoder and the column decoder are grouped in a pair to correspond to one of the sectors.

26. The NOR flash memory device as set forth in claim 25, wherein the column selector includes a global column decoder corresponding to each one of the banks.

* * * * *