宇宙科学分野のビッグデータ処理を高速化する 自然数インデックス (NNI: Natural Number Index)

古庄 晋二[*1]，飯沢 篤志[*2]，長尾 正[*3]，山本 幸生[*4]
早部 秀一[*1]，生座本 義勝[*1]，小林 正英[*1]

Accelerating Big Data Processing in Space Sciences with Natural Number Index (NNI)

FURUSHO Shinji[*1], IIZAWA Atsushi[*2], NAGAO Tadashi[*3], YAMAMOTO Yukio[*4]
HAYABE Shuichi[*1], OZAMOTO Yoshikatsu[*1], KOBAYASHI Masahide[*1]

ABSTRACT

In the field of space science, there is a need to combine and analyze huge amounts of data such as observation data and other big data on a daily basis. Most of them are atypical processes that we cannot afford to use much time for tunings, such as index design and others. In addition, their processing time tends to be long and needs to be reduced significantly. Therefore, to meet such needs, we propose to utilize the Natural Number Index (NNI), which can process tabular data at high speed in a very wide range of cases. Any tabular data can be uniquely decomposed into the components defined by the NNI. Then, using those components, it becomes possible to design the set of algorithms to access the order relationships intrinsic in the tabular data. While existing indexes are single- purpose, utilizing an external data structure apart from the data to be processed, the NNI is a multi- purpose index utilizing internal order relationships intrinsic in the tabular data. For tabular data, the NNI can (1) process any part of the data at high speed, and (2) speed up a variety of processes, including relational algebra operations. (3) In many cases, individual processing is orders of magnitude faster than existing indexes. (4) It has a mechanism to efficiently cascade individual processes to create compound processes. Thus, the NNI can achieve general-purpose, fast data processing without tuning. Thus, it can greatly improve the efficiency of various big data processing in the space science field, both at design time and at runtime. By using the NNI, we can expect to accelerate various researches in space science.

Keywords: Natural Number Index, Big Data, Relational Algebra, Space Science Field, Apollo.

---

[*] 2020 年 12 月 4 日受付（Received December 4, 2020）
[*1] 株式会社エスペラントシステム（Esperant System Co., Ltd.）
[*2] リコー IT ソリューションズ株式会社（RICOH IT Solutions Co., Ltd.）
[*3] Layman's Admin
[*4] 宇宙航空研究開発機構 宇宙科学研究所（Institute of Space and Astronautical Science, Japan Aerospace Exploration Agency）

This document is provided by JAXA.

概　要

宇宙科学分野では観測データなど膨大なデータが存在し、それらのビッグデータを組み合わせて解析するニーズが日常的に存在する。その多くはインデックス設計などのチューニングにいちいち時間をかけられない非定型処理である。またそれらの処理時間は長大になりがちで大幅に短縮する必要がある。そこで著者らはこのようなニーズに応えるため、表形式データを多様なケースで高速処理できる、自然数インデックス（NNI: Natural Number Index）の利用を提案する。全ての表形式データは NNI が定める成分に一意に成分分解できる。するとそれらの成分を介して、表形式データに内在する順序関係を使う多様なアルゴリズム群が設計可能になる。既存のインデックスは処理対象データの外部にあるデータ構造を利用し単用途であるが、NNI は処理対象データに内在する順序関係を利用する多用途のインデックスである。NNI は、表形式データに対して、① データのどの部分でも高速処理できる。② 関係代数演算を含む多様な処理を高速化できる。③ 個々の処理が既存のインデックスより桁違いに高速であることが多い。④ 個々の処理を効率的にカスケードして複合処理を作る仕組みを持つ。このため NNI はチューニングレスで汎用的な高速データ処理を実現できる。従って宇宙科学分野のさまざまなビッグデータ処理を設計時と実行時の両方で大幅に効率化できる。つまり NNI は日々のビッグデータの非定型処理を容易にし、加えてこれまでやれなかった処理も実行可能にする。NNI の活用で宇宙科学分野の様々な研究の加速が期待できる。

キーワード：自然数インデックス、ビッグデータ、関係代数、宇宙科学分野、アポロ

1　はじめに

宇宙科学分野では観測データだけでは無く、衛星の運用管理データなどの多様な周辺データも膨大である。これらに他の分野から得られたデータを組み合わせてデータ処理をしたいというケースも日常的に存在する。加えて通信速度の急激な向上により宇宙科学分野、それ以外の分野のビッグデータもダウンロード可能な時代が来る。これらのビッグデータを組み合わせて 2 次的、3 次的なビッグデータも生成されてゆく。

このような状況の中、宇宙科学分野におけるデータ処理においては、以下に示す要求が存在していると考えている。

1. **データ結合の容易性**：観測データは適当な時間間隔で分割され、HDF5 形式などのデータとして提供されている。データを利用する時にはこれらの中から必要なものを選んで結合し、一つの表データとして扱えるとデータ処理がしやすくなる。分割されたデータを容易に結合できることが、その後のデータ処理をしやすくする基礎となる。
2. **検索の高速性**：データ全体は膨大であるが、一時に処理するデータは、その一部である。通常は、時系列の範囲や特定の項目の値（例えば機種名やフラグ）を指定して検索し、適当な個数（たとえば、1 万件とか 10 万件とか）のデータを抽出してきて、データ処理をし、分析する。必要な時に、それに適した条件を指定して、適切なデータを抽出する処理が高速にできると、抽出後に本来やるべきデータ処理・分析に集中できるようになる。このようにして選択されたデータはそれだけを別テーブルに格納したり、CSV に出力する場合もある。

This document is provided by JAXA.

3. **ソートの高速性**：データをある項目をキーとして並べ替える処理は頻繁に発生する。例えば、時系列データは時系列の順で格納されているとは限らず、時系列順にソートする必要があることがある。複数の項目をキーとすることも多い。検索して抽出されたデータをある項目でソートすることも多い。
4. **データの組み合わせ操作の必要性**：キャリブレーションや衛星データと地上面データの関連分析のために同時刻に発生した 2 系統のデータを 1 レコードに並べた上で、処理・分析を進めることがある。そのためには、リレーショナルデータベースで提供されるジョイン操作が必要である。分析のためのデータを作る際に、マスタテーブルなどから付加情報を補充したい場合があるが、そこでもジョイン操作が使われる。
5. **柔軟なデータ処理**：ジョインした上で系統間での数値の差を算出し、その差を集計したり、ソートしたりすることが必要である。異常値についてはそのレコード群を抽出して取り出すことも必要になる。抽出したレコード群を修正し、元に戻すが必要になる。そしてこれらの処理は対話型にもバッチ処理にも行えなければならない。対話型処理の際には、データをその場で表示する機能も必要である。

観測データをはじめとする多様なビッグデータについてこのようなニーズに応えるためには、ビッグデータを事前設計なしに組み合わせて使う、つまり、表計算のように多様な操作ができ、リレーショナルデータベースのようにジョインやマッチングが行える仕組みが必要だと考える。

その仕組みでは、すべてのカラムの取り合わせ、すべての部分集合を等しく高速化できることが求められる。例えば多数の次元を持つ集計を行う場合や、複数のカラムをジョインキーとしてジョインする場合に対応するためには、単一のカラムではなく任意のカラムの組合せに対してもソートできなければならない。検索などで絞り込んだレコード群（部分集合）に対しても、集計やジョインを行う場合があり、それらに対応できなければならない。また、その仕組みには表形式のビッグデータを効率的に処理する上で欠かせない関係代数演算をサポートすることが求められる。

このような仕組みがあれば、様々な目的を持った観測データの利用者が、様々なデータを様々に組み合わせてその目的に合ったデータを作り出すことが可能になる。

この仕組みを実現するには、リレーショナルデータベースシステムを利用することが考えられる。既存のリレーショナルデータベースシステムなどでは、事前に用途を想定してインデックスを付加することで高速化を図る。リレーショナルデータベースシステムでは、データの挿入、削除が頻繁に生じる環境での検索の効率、特に範囲検索の効率を考えて B+木を利用したインデックス構造を利用することが多い [1) 2)]。しかし、インデックスのように処理対象のビッグデータの外部に別のデータ構造を構築して高速化を図る限り上記のようなすべてのケースに対する操作を高速化することは困難である。

そこで、我々が以前から提案している、表形式データを幅広いケースで高速処理できる**自然数インデックス**（NNI: Natural Number Index）の利用を提案する。NNI は表形式のビッグデータの編集・加工・分析を目的とするインデックスで単一のコンピュータ上かつインメモリで運用される。既存のインデックスは処理対象データの外部にあるデータ構造を利用し、それを本質的には一つのアルゴリズムでアクセスする単用途である。NNI は処理対象データに内在する順序関係を利用し、多くのアルゴリズム群で

This document is provided by JAXA.

アクセスする多用途のインデックスである。

NNI の仕組みを述べる。全ての表形式データは NNI が定める成分に一意に成分分解できる。その成分は表形式データに内在する順序関係を表している。するとそれらの成分を使って順序関係を操作するアルゴリズム群が設計可能になる。NNI の実体はそのアルゴリズム群である。

① NNI のアルゴリズム群はすべてのカラム、すべてのカラムの取り合わせ、すべての部分集合に使用できる。表形式データに内在する順序関係が、すべてのカラム、すべてのカラムの取り合わせ、すべての部分集合に等しく存在するためである。
② NNI のアルゴリズム群は検索・集計・ソート・計算・データ変換およびビッグデータを効率的に処理する上で重要なジョインなどの関係代数演算、他に使用できる。これらの処理がすべて順序関係の操作であるためである。
③ NNI の各アルゴリズムは既存のインデックスを使用する場合より桁違いに高速であることが多い。それは本論文中のベンチマークで示される。
④ NNI の各アルゴリズムを効率的にカスケードして多様な複合処理を構成できる。各アルゴリズムの入力と出力が成分（順序関係）であるためである。
⑤ NNI のアルゴリズム群は更新直後の表形式データに直ちに適用できる。データ更新時に成分（順序関係）が自動的に更新されるためである。新たにジョインや集計操作などによって表形式データを作成する場合も同様で、NNI は作成直後の表形式データに直ちに適用できる。

つまり、NNI は表形式データの①全域をカバーでき、②多様な処理を行え、③個々のアルゴリズムが高速で、④効率的な複合処理が構成でき、⑤データの更新・生成後でも直ちに使用できる。これらは外部構造を作らずにデータそのものに内在する順序関係を使う NNI ならではの特長である。

NNI を使えば手間暇のかかるデータベース設計作業が不要になりこれまで着手できなかった非定型処理が可能になる。NNI を使えばデータベース処理が高速になり時間的制約で行えなかった処理が可能になる。つまり NNI を使えば宇宙科学分野のさまざまなビッグデータ処理が設計時と実行時の両方で大幅に効率化され、実行可能になる。NNI の活用で宇宙科学分野の様々な研究の加速が期待できる。

This document is provided by JAXA.

2　自然数インデックス（NNI: Natural Number Index）

本論文では特に断らない限り単一のコンピュータ上の表形式データを前提として説明する。表形式データはいくつかの成分に一意に成分分解できる。表形式データの構造がレコードの 1 次元配列であることに由来して、これらの成分は全て 1 次元の配列である。

後述する SVL という 1 種類の成分を除き、他の全ての成分は自然数（ただし 0 から始まる）を要素とする配列である。SVL のみは自然数ではなく、整数、浮動小数点、文字列など大小関係を一意に定めることができるスカラー型のデータを要素にする配列である。

この成分分解を行うと表形式データに内在する順序関係へのアクセスが可能になり、それらを使うアルゴリズム群が設計可能になる。NNI の実体はそのアルゴリズム群である。

NNI のアルゴリズム群は成分を通じて表形式データにアクセスする。SVL を除く全ての成分は自然数を要素とすることと、多くのアルゴリズム（例えば後述するソート）は、しばしばその SVL を使用しない。「自然数インデックス」との命名はここから来ている。

さて、NNI は 2 種類の成分分解を使う。第 1 が実テーブルの成分分解で 3 種の成分を持つ。第 2 が仮想ジョインテーブルの成分分解で 5 種の成分を持つ。基本となるのは実テーブルの成分分解である。これらは後ほど本論文中で説明される。（なお、仮想ジョインテーブルとは成分分解された 2 つの実テーブルを組み合わせて仮想的に生成されるテーブルである。）

表形式データの成分分解は 2 種類あるが、基本は実テーブルの成分分解である。そこで解説は実テーブルの成分分解に基づいて進め、仮想ジョインテーブルの成分分解はそのアルゴリズム群の説明の際でのみ触れることにする。

2.1　実テーブルの NNI の構成要素

NNI では、表形式データ（以後、テーブルと呼ぶ）をカラムごとのデータ列に分割し、それぞれのカラムデータを以下に挙げる 2 種類の配列に分割する。以降、これを成分分解と呼ぶ。

- SVL (Sorted Value List)
- NNC (Natural Numbered Column)

SVL はカラムの出現値を、①重複無く、②昇順に並べた配列である。NNC はカラムの各値を、SVL 中のその値の格納位置（自然数）に置き換えて表した配列である。従って、その各要素の値は、高々、SVL の配列サイズを超えない。SVL と NNC を合せて、自然数インデックス F (NNIF: Natural Number Indexed Field) と呼ぶ。

NNI は、数学で言う「順序集合」を扱うための配列である、OrdSet（Ordered Set）を使う。OrdSet は、レコード番号（NNC 内の位置）を並べて作る。OrdSet は集合なので、重複する要素は出現しない。各要素の値は、NNC のサイズを超えない。

OrdSet を用いると、検索でヒットしたレコード群を記録することができる。また、ソートで並べ替えたレコード群の順序を記録することができる。

検索やソートを行う度、新たな OrdSet が作られる。OrdSet を切り替えて、検索やソートなどの処理を

This document is provided by JAXA.

行う前のテーブルも、行った後のテーブルも読み出すことができる。

2.2 NNI が高速な理由

NNI は以下の理由で高速処理が可能である。

- カウンティングソート[4] など、これまで整数データでしか成り立たなかった高速アルゴリズムが文字列、浮動小数点データに対しても使える。
- 必要な成分のみを選択的に使って処理を完了できる。
  例えば、ソートでは SVL を使わない。このためメモリアクセス量が減り、高速になる。
- 不変な成分を共有もしくはコピーにより使い回せる。
  検索やソートなどでは NNC と SVL は処理前と処理後で不変である。そのため処理前後で不変成分を共有でき、これらを生成する手間が要らない。
  集計では、集計元テーブルの次元に指定された SVL は、コピーしてそのまま集計結果テーブルの次元の SVL にでき、これらを生成する手間が要らない。
- 処理をカスケードできる。
  （検索・ソート・集計などの）処理結果は成分分解された状態で出力される。そのため処理をカスケードできる。複雑な処理を単純な処理のカスケードで実現できるため、高速である。（単純な処理は複雑な処理よりも高速である。）
- マルチコアを使った並列処理アルゴリズムが処理の様々な局面で使える。

さらに、以下の点も高速化に寄与する。

- 整数の操作が多く、文字列操作が少ないため、CPU が効率的に処理できる。
- 多様な処理を高速化できるので、ボトルネックを生じにくい。
- メモリアクセスの局所性が高い。またシーケンシャルなメモリアクセスが多い。
- 省メモリで高速化に有利なインメモリにしやすい。
  - 複数のテーブル間で成分が共有され、重複保持されない。
  - 整数（自然数）化されていて、メモリ消費量が少ない。
  - 既存のインデックスのように付加的なメモリを必要としない。

2.3 NNI と処理の安定性

アルゴリズムを解説する前に、「安定性」について説明する必要がある。NNI の大きな特徴として、各処理の出力が成分分解された状態で出力され、高速性を保ったまま処理をカスケードできることは既に述べた。カスケードを有意義に行う際に必要なのが安定性である。安定性とは、検索やソートなどの処理の目的を害さない範囲で、処理前のレコード順を保存する性質である。安定性が無いとき、検索やソートの出力は 1 通りに定まらず、多数ある。安定性があるとき、検索やソートの出力はただ 1 通りしかない。

NNI の実際の実装形態では、検索、ソートのアルゴリズムは常に安定性を保証する。Distinct や集合演算では処理速度は多少遅くなるが安定性をもつアルゴリズムを使用するか、処理速度を優先して安定性を求めないアルゴリズムを使用するか、API のパラメータで指定する。

このように安定性を重要視するのは、安定性が処理のカスケード接続を有意義にするか、無意味にするかを左右するためである。

This document is provided by JAXA.

ソートを例に述べよう。性別と年齢の 2 つのカラムを持つテーブルを、性別、年齢別にソートしたいとする。ソートの安定性があれば、まず年齢別にソートし、次に性別でソートすれば良い。もし、安定性がなければ、年齢別にソートした後、それを性別でソートすると、年齢別のソート順が破壊され目的とするソート結果が得られない。

2.4　NNI の説明で扱う記法

NNI は 3 種類の配列を使用する。その表記法を述べる。

- カラムに関係する 2 つの配列（SVL, NNC）の表記
  これらの配列はテーブルのカラムに所属する。
  最も正式な表記は、配列名の右肩に (“テーブル名”-“カラム名”) である。
  どのテーブルに所属するかは自明である場合は、右肩には (“カラム名”) のみを書く。
  正式な表記例：　SVL $^{(\text{TableA-Item1})}$
  略式な表記例：　SVL $^{(\text{Item1})}$
- 集合に関係する配列（OrdSet）の表記
  これらの配列はテーブルに所属する。かつ、検索やソート、集合演算などを行うと次々に生成される。生成される度に識別番号が振られる。
  最も正式な表記は、配列名の右肩に (“テーブル名”-識別番号) である。
  通常、どのテーブルに所属するかは自明であるので、右肩には (識別番号) のみを書く。
  正式な表記例：　OrdSet $^{(\text{TableA-0})}$
  略式な表記例：　OrdSet $^{(0)}$
- サブテーブルの表記
  検索やソートなどの処理をする前後のテーブルは、データ本体（NNC/SVL）は同一なので上記の OrdSet らを切り替えることで表現できる。
  特定の OrdSet との組合せで表現されたテーブルを「サブテーブル」と呼ぶ。
  サブテーブルを表記したいとき、OrdSet らの右肩にある識別番号をそのままテーブル名の右肩に載せて表記する。
  表記例：　TableA $^{(0)}$
- レコードの表記 1
  レコードを各カラムの値の集合と見なし、（ カラム 1 の値, カラム 2 の値, ... ）　と表記することがある。
- レコードの表記 2
  テーブルを、0 から始まる自然数で特定されるレコードが、その自然数の順番で並べられた配列と見なし、以下で、レコードを表すことがある。
  表記例：　TableA [i]　...　i 番目のレコード
- レコードの結合を表す表記
  レコードの表記 1 および 2 を使って、レコードの結合を表すには、&記号を用いる。
  表記例 1：　(Jack, 20) & (Betty, 19)
  表記例 2：　TableA[3] & TableB[1]

This document is provided by JAXA.

- レコード番号の表記
  仮想ジョインテーブルの説明には、ジョインテーブル上でのレコード番号、Master 側テーブルのレコード番号、Slave 側テーブルのレコード番号の 3 つのレコード番号が必要になる。スカラーとしてのレコード番号は以下のように表記する。
  ジョインテーブルのレコード番号： $I_{(join)}(i)$ もしくは $I_{(join)}$
  MASTER 側テーブルのレコード番号： $I_{(master)}(i)$ もしくは $I_{(master)}$
  SLAVE 側テーブルのレコード番号： $I_{(slave)}(i)$ もしくは $I_{(slave)}$
- 図中の配列の表記について注意
  図中で、新規に作成された配列、更新された配列は、太字・太枠にしている。
  図中で配列中の要素の出自を示すため、カラム毎に割り振られた色で塗りつぶすことがある。

NNI は配列を多用するが、以下に留意されたい。

- すべての配列は 0 ベースである。
- A が自然数を要素とする配列であるとき、A[-1] は 0 と定義する。

2.5 成分分解

図 1 を用いて成分分解を説明する。表示データの“Name”カラム { Bob, Alice, Beth, Cathy } は $NNC^{(Name)}$ および $SVL^{(Name)}$ に分解されている。“Age”カラムも同様である。これを成分分解と呼ぶ。

各カラムを成分分解した後、$OrdSet^{(0)}$ を付加すると成分分解が完了する。$OrdSet^{(0)}$ は、Root OrdSet とも呼ばれる特別な OrdSet で、サイズは NNC のサイズと等しく、その要素は 0 からの連番である。

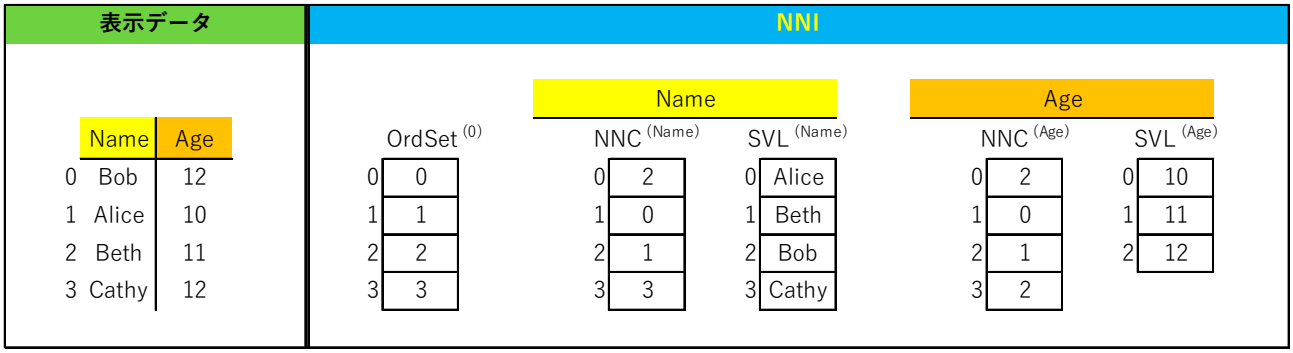


図 1. 成分分解

CSV などで与えられたテーブルを NNI の形式に変換できることは明らかである。また、NNI の形式から表示データを作ることも簡単にできる。

さて、テーブルの加工編集、ジョインなどにより SVL 中に NNC から参照されない値が挿入されている NNIF が出現することがある。その場合でも、データを読み出したり、処理したりすることは可能であるので、あまり支障が無い。通常、配列群をディスクにセーブするタイミングで、参照されない値を外す（この処理をコンデンスと呼ぶ）。

This document is provided by JAXA.

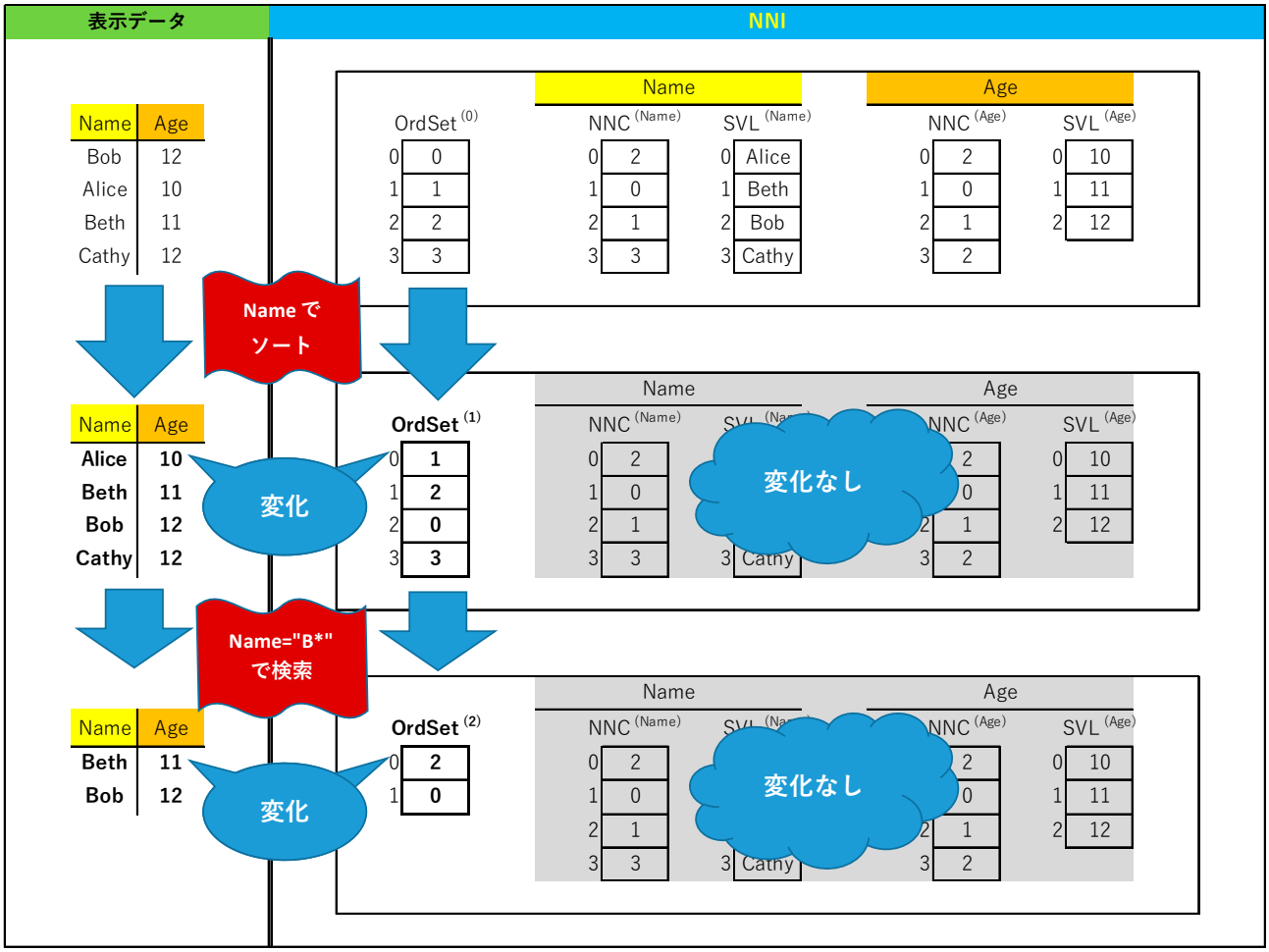


図 2.　ソートと検索による OrdSet の変化

2.6　OrdSet について

図 2 では“Name”カラムでソートし、検索したとき OrdSet がどのように変化するかを示す。

図 2 で示すように NNIF は変化しない。従って、ソートや検索の結果は、OrdSet $^{(1)}$ や OrdSet $^{(2)}$ を記録するだけで保持できる。処理の高速化と省メモリの両方に役立つ。

2.7　データの読み出し

図 3 にデータの読み出し方法を示す。図 3 では、Name でソートし、検索した結果生成された一番下段のテーブルの一番下のレコードを読み出している。

一般的には以下の方法でアクセスする。図 3 の場合は、$i = 2, j = 1$ を選択している。

$$\text{Name の値} = \mathrm{SVL}^{(\mathrm{Name})}[\,\mathrm{NNC}^{(\mathrm{Name})}[\,\mathrm{OrdSet}^{(i)}[\,j\,]\,]\,]$$
$$\text{Age の値} = \mathrm{SVL}^{(\mathrm{Age})}[\,\mathrm{NNC}^{(\mathrm{Age})}[\,\mathrm{OrdSet}^{(i)}[\,j\,]\,]\,]$$

This document is provided by JAXA.

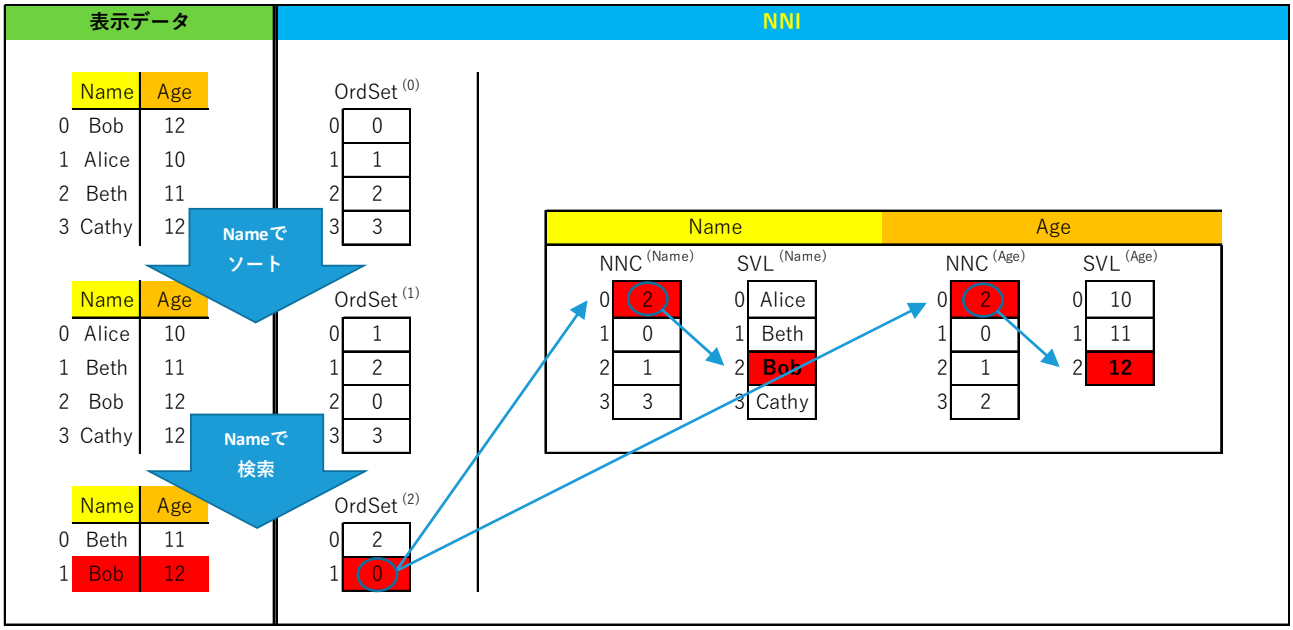


図 3. データの読み出し

2.8 成分分解の実施方法

CSV データなどのデータを読み込んで、一気に NNC と SVL を構成する、成分分解の効率的な方法が知られている。[15)]

3 実テーブル上のアルゴリズム（検索・ソート・集計）

3.1 実テーブルの検索のアルゴリズム [12)]

NNI の検索は主キーのように 1 つしか存在しない値を検索することもできるし、範囲の検索により 100%ヒットするケースの検索もできる。値群を指定して検索することもできる。複数項目の条件で検索する場合は、別途集合演算を必要とし、説明が煩雑になるのでここでは単一のカラムでの検索に限定して説明を進める。

NNI の検索処理の時間は、数件しかヒットしない場合も、全件がヒットする場合も、大差が無い。そのため、ヒットさせる件数が小さい場合は、既存のインデックスに比べて遅い。ヒットする件数が多いとき、既存のインデックスに比べて大幅に速い。

NNI の検索には安定性がある。この性質により検索やソートなどを有意義にカスケードすることが可能になる。

また、OrdSet を分割し、各コアに割り当てることで簡単に並列処理ができる。

3.1.1 アルゴリズムの概要

検索は、2 ステップで行う。

1. フラグ配列の作成

   SVL と同じサイズのフラグ配列（Flags とする)を用意する。SVL の各値が検索条件に合致しているかどうかをフラグ配列に書き込む。

This document is provided by JAXA.

検索の条件が範囲で指定されるなら、SVL の値が条件を満たす範囲に対応するフラグ配列の範囲を連続的にマークすれば良く、高速になる。

値群で検索するときは、その値群を成分分解することでその値群の SVL を作成し、その SVL と検索対象カラムの SVL とマッチングを取ることで高速にフラグ配列をマークすることができる。

2. ヒットテスト

Flags[NNC[$OrdSet^{(k)}$ [i]]] を見て、マークされていたら、$OrdSet^{(k)}$ [i] を検索結果集合 $OrdSet^{(k+1)}$ に追加する。

（詳細を付録 1 に記載する。）

3.1.2　アルゴリズムの特性

高速性：

SVL のサイズが NNC のサイズに比べて小さいとき、ヒットテストの時間が主になる。整数のシーケンシャルアクセスを主体とするアルゴリズムで、高速に多数のヒットレコード番号を取り出すことができる。ステップ数は、検索前の OrdSet のサイズを s として、O(s) となる。

順序の安定性：

また検索の安定性を有し、他の検索やソートなどの処理と有意義にカスケード接続できる。

並列性：

OrdSet を分割し、CPU の複数のコアに割り当てることで、容易に並列処理できる。

3.2　実テーブルのソートのアルゴリズム

通常、NNI のソートは、カウンティングソートで行う。カウンティングソートはソート対象の OrdSet のサイズを s として、O(s)で実行でき、ソートの安定性も有する。マルチコアで並列にソートを行う技術も知られている。[2) 19)]

優れたソートであるがこれまであまり使われなかったのは、以下の理由による。

- 整数にしか使用できない。
- カーディナリティが高いと効率が低下する。

NNI ならデータは自然数化されて保持されているし、カーディナリティは高々SVL のサイズで済み、これらの問題は発生しない。

ソートに安定性があるので、検索及びソートを有意義にカスケードすることができる。

3.2.1　アルゴリズムの概要

ソートは、3 ステップで行う。

1. カウントアップ

SVL と同じサイズのカウント配列に、SVL の各値が何個出現するかを記録する。

2. 累計数化

カウント配列を累計数配列化する。

3. 転送

ソート元 OrdSet の各値を、ソート結果の OrdSet 上に転送する。

（詳細を付録 2 に記載する。）

This document is provided by JAXA.

3.2.2 アルゴリズムの特性

高速性：

OrdSet のサイズを r、SVL のサイズを s とする。カウントアップを行う第一段階で O(r)、累計数を作成する第二段階で O(s)、転送を行う第三段階で O(r) のステップを必要とする。

従って、処理時間は OrdSet のサイズにほぼ比例するだけであり、高速である。

順序の安定性：

安定である。

並列性：

可能である。[19)]

3.3 実テーブルの集計のアルゴリズム

これまで、検索やソートについて説明してきた。これらの操作は OrdSet だけで規定できるサブテーブルしか生成しない。一方、集計結果テーブルを生成する「集計」操作は実テーブルを生成する。

集計結果テーブルの次元を作成する際は、集計元テーブルの次元の SVL をそのまま流用できる。そのため高速である。集計結果テーブルの測度は新たに成分分解して作る。

集計の作成は、キューブ方式とソート方式の 2 つの方法がある。NNI は状況に応じてこの 2 者を使い分ける。キューブ方式は高速である一方、キューブサイズが膨大になると実行できない。ソート方式はそのような制約がない。アルゴリズムの説明はソート方式で行う。

3.3.1 アルゴリズムの概要

集計結果テーブルの作成は以下の方法で行う。

1. 集計元テーブルの次元群でソートする。
2. ソートされた次元群の NNC が同一な範囲をグループ化して集約する。グループ毎に測度を算出する。
3. 測度を成分分解する。

（詳細を付録 3 に記載する。）

3.3.2 アルゴリズムの特性

高速性：

OrdSet のサイズを s、集計結果のサイズ（レコード数）を r とする。次元でソートを行う第一段階で O(s)、次元と測度を作成する第二段階で O(s)、測度を成分分解する第三段階で O(r * log(r)) のステップを必要とする。

通常、r が小さいので処理時間は O(s) になる。

並列性：

各ステップはマルチコアを使った並列処理でできる。

This document is provided by JAXA.

4　実テーブル上のアルゴリズム（一括変換関連）

4.1　SVL の共通化のアルゴリズム

SVL はそれぞれのカラム毎に独立に存在している。そのためカラム毎に NNC の保持する値は異なる意味を持つ。例えば UNION を行う際、二つの NNC は 1 つになり、SVL も一つになる。この操作を行うには、最初に SVL を 1 つにし、その後、各 NNC を調整する（UNION の場合はそのまま連結する）。

この処理は、「SVL の共通化」と呼ばれる処理で、UNION に限らず、ジョインや複数データの一括更新などでも出現する。

この処理は、配列をその先頭から最後に向けてアクセスするシーケンシャル処理が主体で高速である。

4.1.1　アルゴリズムの概要

2 つのステップで成り立つ。

1. newSVL の作成
   二つの SVL の各要素を先頭から見比べながら、小さい要素を newSVL に格納する。同時に newSVL に格納した位置を変換配列（Conv 配列）に記録する。
2. NNC の変換
   各 NNC を上記の Conv 配列を参照しながら変換する。

（詳細を付録 4 に記載する。）

4.1.2　アルゴリズムの特性

高速性：

SVL のサイズが NNC のサイズより小さいと仮定する。2 つの NNC のサイズを n1、n2 とするとき O(n1 + n2) となる。

並列性：

主要な各ステップはマルチコアを使った並列処理ができる。

4.2　実テーブルの UNION のアルゴリズム

NNI の場合、サブテーブル間でも UNION できる。UNION の処理は SVL の共通化処理を含む。この SVL の共通化処理はシーケンシャルかつ SVL の格納値の比較以外は整数しか比較しないため、高速である。

4.2.1　アルゴリズムの概要

UNION の対象となる NNIF ペア毎に NNIF の合併処理を行う。

（詳細を付録 5 に記載する。）

4.2.2　アルゴリズムの特性

高速性：

UNION を行う 2 つのサブテーブルの NNC のサイズを n1、n2 とし、UNION で生成されるテーブルのカラムの数を c とするとき O(c * (n1 + n2)) となる。

This document is provided by JAXA.

並列性：

主要な各ステップはマルチコアを使った並列処理ができる。

5 実テーブル上のアルゴリズム（データ更新関連）

5.1 実テーブル上の複数のデータの一括上書きのアルゴリズム

NNI では、1 つのデータを更新するときと、多数のデータを更新するときとの、処理時間の差が少ない。そのため、更新はできる限りバッファに蓄積しておき、検索や集計などの処理が要求されたときに更新をコミットする。こうすることで、多くのデータベースシステムに比べても高速な更新が可能になる。

データの上書きのアルゴリズムは、被上書きデータはもちろん、上書きデータそのものも成分分解した上で上書きを行うことで、効率的になっている。

5.1.1 アルゴリズムの概要

以下の方法で行う。

1. 上書きデータ群を成分分解し、NNIF（NNC と SVL）にする。
2. 上記の NNIF と、被上書き NNIF の間で、SVL の共通化を行う。（付録 4.1 参照）
3. 上書きデータの NNC を被上書きデータの NNC 上に書き込む。

（詳細を付録 6 に記載する。）

5.1.2 アルゴリズムの特性

高速性：

更新データの SVL のサイズを s1、被更新データの SVL のサイズを s0 とする。

更新データのサイズを m とし、被更新データのサイズを n とする。

新 SVL を作成する段階で O(s1+s0)、NNC を更新する段階で O(m+n) のステップとなる。

処理時間は被更新データのサイズにほぼ比例し、高速である。

並列性：

主要な各ステップはマルチコアを使った並列処理ができる。

5.2 実テーブル上のレコード群の挿入のアルゴリズム

NNIF のレコード群挿入は、表計算の複数行の挿入に相当する処理である。全カラムの NNIF を更新するとともに、ルート OrdSet（=$OrdSet^{(0)}$ ）も更新される。

5.2.1 アルゴリズムの概要

以下の方法で行う。

1. カラム毎に以下を行う。

   SVL の最初の要素が null でなければ、SVL の先頭に null を挿入し、それに伴い、NNC の更新を行う。（そのカラムのデータ型の最小値を null と見立てる。）

This document is provided by JAXA.

2. カラム毎に以下を行う。
   NNC のレコード群挿入位置に、挿入レコードの数だけ、0 を挿入する。
3. OrdSet$^{(0)}$ のサイズを挿入レコードの数だけ増やす。

（詳細を付録 7 に記載する。）

5.2.2　アルゴリズムの特性

高速性：

挿入レコード数を n1 とし、被挿入側のレコード数を n0 とするとき、O(n1 + n0) となる。

並列性：

主要な各ステップはマルチコアを使った並列処理ができる。（各カラムの処理を複数のコアで分担できる。）

5.3　実テーブル上のレコード群の削除のアルゴリズム

NNIF のレコード群削除は、表計算の複数行の削除に相当する処理である。全カラムの NNC を更新するとともに、すべての OrdSet が更新される。

5.3.1　アルゴリズムの概要

以下の方法で行う。

1. カラム毎に以下を行う。
   NNC のレコード群削除位置から、削除レコードの数だけ、要素を削除する。
2. すべての OrdSet を更新する。

（詳細を付録 8 に記載する。）

5.3.2　特性

高速性：

NNC のサイズを n とするとき、O(n) となる。

並列性：

主要な各ステップはマルチコアを使った並列処理ができる。（各カラムの処理を複数のコアで分担できる。）

6　仮想ジョインテーブル

成分分解されている 2 つの実テーブル（サブテーブルでも良い）を仮想的にジョインして仮想ジョインテーブルを作ることができる。ジョインキーは複数であってもよく、インナージョインもしくはフルアウタージョインのいずれでも可能である。

仮想ジョインテーブルは、Master 側と Slave 側の 2 つのテーブルからの写像として合成される。仮想ジョインテーブルは、仮想ジョインテーブルのレコード番号を指定すると、そこから Master 側および Slave 側テーブルのレコード番号を直ちに特定する仕組みを備える。このことにより、仮想ジョインテー

This document is provided by JAXA.

ブルのレコードはMaster側とSlave側のレコードを参照して取得・表示できる。参照先を書き換えることは影響範囲が大きく難しいことから、仮想ジョインテーブルはデータの加工・編集はできない。その一方、検索・ソート・集計は高速にできる。実テーブルではメモリ不足で運用が困難な20億レコードでも（写像であるため）運用できる。

仮想ジョインテーブルの処理は実テーブルよりも遙かに高速なことが多い。例を挙げよう。Master側が1000万レコード、Slave側が100万レコード、仮想ジョインテーブルが20億レコードであったとする。Master側の1レコードに平均200のSlave側のレコードが結合されていることになる。ソートは、Master側の1000万レコードをソートすれば、仮想ジョインテーブル上では20億レコードをソートしたことになる。当然高速である。集計はMaster側とSlave側それぞれを集計して後に説明する方法でその積を作ることで達成できる。20億レコードにアクセスして集計する方法と、1000万レコードの集計と100万レコードの集計の積を作る方法のどちらが高速かは自明である。

仮想ジョインテーブルの典型的な用途は以下である。

1. 検索と集計の組合せでの使用
   機械装置の出荷記録テーブルがあるとする。また、各機械装置の構成部品テーブルがあるとする。この2つをジョインすると出荷部品の一覧テーブルができる。出荷期間で「検索」し、その期間にどの部品が何個出荷されたことになるかを「集計」する。
2. 検索とソートの組合せでの使用
   上記の出荷部品の一覧テーブルを使って、特定の不良が発見された部品を含む機械装置を「検索」する。その上で、出荷先毎にソートし、出荷先にそれらの機械装置に関するリコールなどの通知を出す。

仮想ジョインテーブルの構造、作成方法、読み出し方法は、かなり複雑なので、本文での説明は割愛し、すべて、付録に譲る。付録9を参照してほしい。

仮想ジョインテーブルの各レコードは、ジョインの元になっている2つのテーブルのレコードをポインターを使わずに、算出により参照している。ポインターは参照数だけ必要だが、算出の場合はジョインキー値毎に1つでよい。このことにより、以下のメリットが得られる。

- 集計などのアルゴリズムを、Master側、Slave側のテーブルの処理に分けて実行しやすい。
- 多彩なアルゴリズムを作りやすい。
- メモリ消費が少ない。
- アルゴリズムの冗長性が減り高速になる。

仮想ジョインテーブルには以下の機能があり、それらを高速に実行できる。

- 検索
- ソート
- 集計
- Slave側のカラムをMaster側に転送
- ジョインにマッチした集合／しなかった集合の取出し
- 仮想ジョインのサブテーブルを実テーブルに変換する

This document is provided by JAXA.

仮想ジョインテーブル上のアルゴリズムの多くは、Master 側のテーブル、Slave 側のテーブルで処理を行い、それをジョインテーブルに反映させるアプローチを取る。こうすることで、効率的な処理が行える。以下、仮想ジョインテーブル上のアルゴリズムについて、その概要と特性を述べる。

6.1　検索のアルゴリズム

6.1.1　アルゴリズムの概要

仮想ジョインテーブルの検索は、Master 側もしくは Slave 側のテーブルを検索して、再度ジョインを行うことで実現する。どちらかのテーブルで検索の後、再度ジョインを行う際には、SVL の共通化とその関連の処理はすでに済んでいるので行わなくて良い。
（詳細を付録 10 に記載する。）

6.1.2　アルゴリズムの特性

高速性：

処理時間は Master もしくは Slave 側の検索対象のテーブルの OrdSet のサイズ (n1, n2 とする) に比例する検索時間と、ジョインテーブルを再作成する時間の和になる。ステップ数は O(n1+n2) である。

安定性：

Master 側のテーブルのカラム間では検索の安定性を有し、他の検索やソートなどの処理と有意義にカスケード接続できる。Slave 側では検索の安定性は保証されない。

並列性：

主要な各ステップは並列処理できる。

6.2　ソートのアルゴリズム

6.2.1　アルゴリズムの概要

仮想ジョインテーブルのソートも検索と同様に、Master 側もしくは Slave 側のテーブルをソートして、再度ジョインを行うことで実現する。再度ジョインを行う際には、SVL の共通化とその関連の処理はすでに済んでいるので行わなくて良い。

ソートキーが 1 カラムのみ、もしくは複数であってもジョインを構成するどちらかのテーブルの一方のみに属している場合は、ソートの安定性を利用してそのジョインの元テーブルをソートした上で、その元テーブルを Master 側にして再度ジョインを行えばよい。

ソートキーが複数であって、ジョインを構成する両方のテーブルに属する場合、通常は、目的とするソートはできない。Slave 側のテーブルのレコード順がジョイン操作によって破壊されるためである。
（詳細を付録 11 に記載する。）

6.2.2　アルゴリズムの特性

高速性：

処理時間は Master もしくは Slave 側のテーブルの OrdSet のサイズに比例するソート時間と、ジョインテーブルを再作成する時間の和になる。ステップ数は Master 側と Slave 側の OrdSet のサイ

This document is provided by JAXA.

ズを n1, n2 として、およそ O(n1 + n2) である。

安定性：

Master 側のテーブルのカラム間ではソートの安定性を有し、他の検索やソートなどの処理と有意義にカスケード接続できる。なお、Slave 側でソートする場合は、Slave 側を Master 側に切り替えてソートを行う。

並列性：

主要な各ステップは並列処理できる。

（詳細を付録 11 に記載する。）

6.3 集計のアルゴリズム

NNI を用いると、ジョインテーブルの集計は特に効率が良いことが多い。ただし、そのアルゴリズムの説明を成分分解した形で行うと煩雑になる。そこで、集計のアルゴリズムの説明はできる限りデータの見かけのイメージを使って行う。

集計方法は大別してキューブ法とソート法がある。ケース毎にどちらが最適かは違ってくる。ここでは、幅広いケースに使えるソート法で話を進める。

6.3.1 アルゴリズムの概要

ジョインキー値毎に分離された各ジョインテーブル内では、各 Master 側レコードにマッチする Slave 側レコード群が等しい。（その逆も成り立ち、各 Slave 側レコードにマッチする Master 側レコード群が等しい。）そこで、ジョインテーブルの集計の基本的なアプローチは、以下となる。

1. ジョインテーブルを、ジョインキー値毎に分離する。分離されたジョインキー値毎のジョインテーブルを集計する。集計結果を順次足し込んで行く。
2. 上記のジョインキー値毎の集計は、Master 側のテーブル、Slave 側のテーブルでそれぞれ集計を行い、その積を作る。積なので効率が高い。

（詳細を付録 12 に記載する。）

6.3.2 アルゴリズムの特性

高速性：

処理時間は Master 側と Slave 側の次元となるカラムでのソート時間と、ジョインテーブルを再作成する時間の和になる。ステップ数は Master 側と Slave 側の OrdSet のサイズを n1, n2 として、およそ O(n1 + n2) である。

並列性：

主要な各ステップは並列処理できる。

6.4 カラム転送のアルゴリズム

6.4.1 アルゴリズムの概要

ジョインテーブルのカラム転送とは、Slave 側のテーブルのカラムを、ジョインキーを経由して Master 側のテーブルにコピーする機能である。転送されるカラムの SVL はそのままコピーすれば良く、NNC

This document is provided by JAXA.

を再計算すれば完了する。高速である。
（詳細を付録 13 に記載する。）

6.4.2　アルゴリズムの特性

高速性：

処理時間は Master 側の OrdSet (i) のサイズを s として、ステップ数は O(s) であり、高速である。

並列性：

主要な各ステップは並列処理できる。

6.5　ジョインにマッチした集合／しなかった集合の取出しのアルゴリズム

6.5.1　アルゴリズムの概要

ジョインテーブルの MAcm / SAcm 及び Master 側／Slave 側の成分群を使用すると、ジョインキーにマッチした集合、しなかった集合を容易に取り出すことができる。Master 側のマッチあり／マッチなしの集合は Master 側のテーブルに新たな OrdSet として追加される。Slave 側も同様である。
（詳細を付録 14 に記載する。）

6.5.2　アルゴリズムの特性

高速性：

Master 側の集合取得の処理時間は Master 側の OrdSet (i) のサイズを sm として、ステップ数は O(sm) であり、高速である。Slave 側はスレイブ側の OrdSet (i) のサイズを ss として、ステップ数は O(sm + ss) であり、高速である。

並列性：

主要な各ステップは並列処理できる。

7　ベンチマーク

NNI を用いた製品である ESPERiC と、ポピュラーな RDB システムである SQLite を比較し、両テクノロジーの特徴を確認する。検索・ソート・UNION・ジョイン・集計・データ更新の各処理の特性の比較を行った。

7.1　測定環境 [2]

1. ハードウェア：　CPU: Core™ i5-3320M / メモリ 16G / SSD 128G
2. OS：　Ubuntu 18.04LTS / Jupyter notebook : 6.0.3

[2] 使用したデータ、SQL 文、測定結果の数値データなどは以下にある。
https://www.kaggle.com/zanjibar/apollo-pse-data-for-benchmark

This document is provided by JAXA.

3. NNI ソフトウェア： ESPERiC-3.0.200120
4. SQLite： 3.32.3（データはすべてメモリ上に展開した上で測定した）

7.2 測定に使用したデータ

使用したデータは、元々JAXA の DARTS にある、Apollo 11 号～17 号が月面で採取した地震計のデータ [20) 3]である。今回、その中から SQLite でも扱える 1 億レコードのデータを選んで使った。

表 1. 測定に使用したデータ

SQL 文で使用したデータ名	詳細
ist_tdz_11	レコード数：1,411,785 カラム 1：名称 datetime / データ型 double / 値の種類数 1,411,322[4] カラム 2：名称 ist / データ型 32bit integer / 値の種類数 1,024 カラム 3：名称 tdz / データ型 32bit integer / 値の種類数 1,024
ist_tdz_16	レコード数：99,614,916 カラム 1：名称 datetime / データ型 double / 値の種類数 99,590,4074[5] カラム 2：名称 ist / データ型 32bit integer / 値の種類数 1,024 カラム 3：名称 tdz / データ型 32bit integer / 値の種類数 1,024
TD_XY_16	レコード数：99,683,514 カラム 1：名称 datetime / データ型 double / 値の種類数 99,578,518[6] カラム 2：名称 tdx / データ型 32bit integer / 値の種類数 1,024 カラム 3：名称 tdy / データ型 32bit integer / 値の種類数 1,024

7.3 CSV の Load 時間の比較

表 2. Load 時間（およびファイルサイズ）

CSV ファイル	NNI	SQLite
TD_XY_16 のデータ 全 2,186,451,045 バイト	TD_XY_16 のデータ 全 2,391,574,756 バイト Load 時間： 38,260 ms	TD_XY_16 のデータ 全 3,860,316,160 バイト Load 時間： 156,950 ms

[3] http://darts.isas.jaxa.jp/planet/seismology/apollo/PSE.html
[4] datetime は一部重複があるがほぼユニークである。
[5] 同上
[6] 同上

This document is provided by JAXA.

7.4　検索の特性比較

使用したデータ：

ist_tdz_16 を使用した。SQLite は datetime カラムにインデックスを張った。

測定と考察：

datetime カラムで検索して、検索ヒット件数を 1～50,022,313 件の範囲で変化させて、検索時間を計った。その結果を図 4 に示す。なお、図中の SQLite の測定結果が 2 通りあるが、計測環境をリブートした後の初回の検索と、リブートを行わない 2 回目の検索とで 2 通りできたものである。図から分かるようにこの 2 つの性能はずいぶん異なる。一方、NNI は 1 回目も 2 回目以降もほぼ同じ特性を示すので、1 回しか測定していない。検索対象カラムは datetime カラムとした。

NNI はヒット件数が 1 レコード～50,022,313 レコードまで変化しても、検索時間は 147 ms ～ 213 ms とさほど変わらない。一方、SQLite は 5.2 ms から 17,024 ms までと大きく変化している。またヒット件数が増えるに従って、1 回目と 2 回目の処理時間は変わらなくなる。キャッシュの効果が失われるためと考えられる。

NNI は検索条件にあまり左右されず、すぐに結果を返す。この特性は表計算のような対話型処理においては、操作者を待たせない（この測定では 0.3 秒以下）快適な操作性に繋がる。

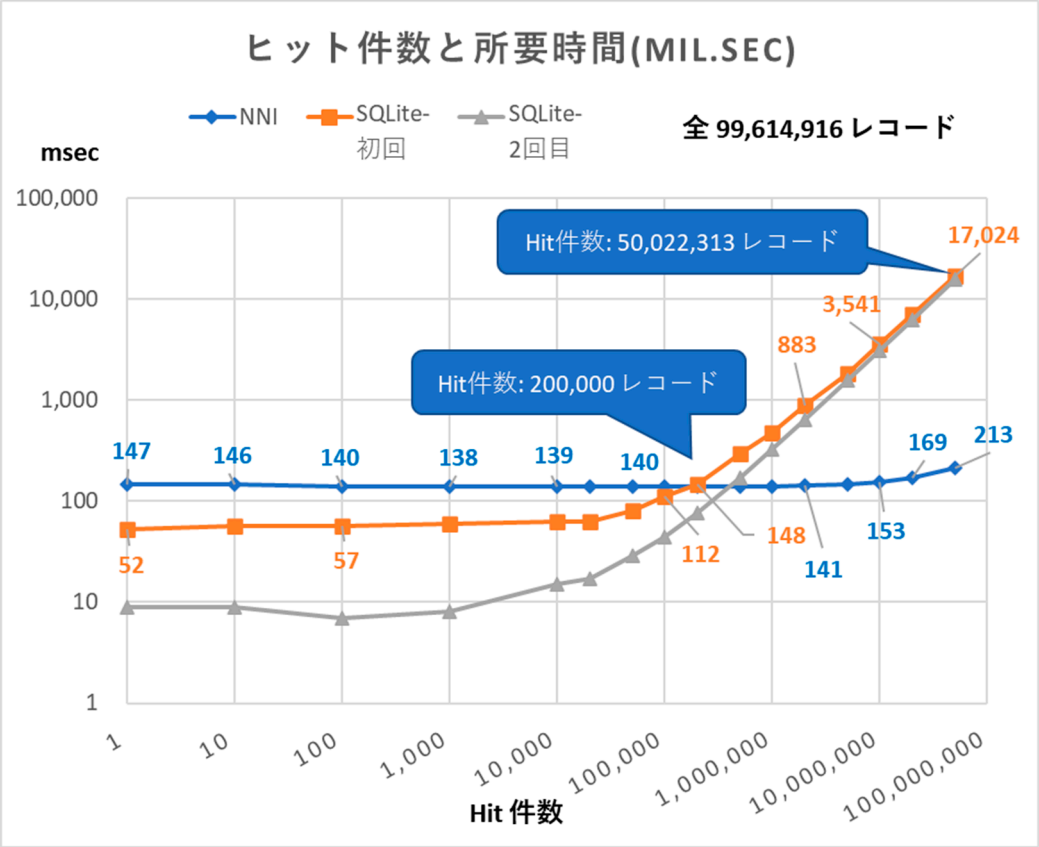


図 4.　NNI と SQLite の検索時間比較（ms）

This document is provided by JAXA.

7.5 ソートの特性比較

使用したデータと測定結果：

ist_tdz_16 を使用した。

NNI ではケースによらず、処理時間のばらつきが少ない。datetime のソートが tdz のソートより時間がかかるのは値の種類数が大きいためである。

SQLite では datetime のソートが tdz のソートより速い。これは datetime が既にほぼソートされた状態であるためと考えられる。

表 3. NNI と SQLite のソート時間比較

処理 (単位 ms)	NNI	SQLite 初回	SQLite 2 回目	SQLite の インデックス
select * from ist_tdz_16 order by datetime;	675	31,160	30,387	datetime にあり
		98,168	80,065	datetime になし
select * from ist_tdz_16 order by tdz;	339	65,044	33,323	tdz にあり
		85,153	61,267	tdz になし
select * from ist_tdz_16 where tdz=0 order by datetime; （4,063,339 件 Hit）	111[7] + 118[8]	3,756	2,976	tdz にあり
		20,009	10,651	tdz になし

7.6 UNION の特性比較

使用したデータと測定結果：

ist_tdz_11 と ist_tdz_16 を使用した。

NNI では値の種類数が大きいとき、SVL の共通化処理のコストが高くなる。表 4 でのパフォーマンスがあまり出ていないのはそのためである。

表 4. NNI と SQLite の UNION 時間比較

処理 (単位 ms)	NNI	SQLite 初回	SQLite 2 回目
create table union_16 as select * from ist_tdz_11 union all select * from ist_tdz_16	6,644	36,435	25,517

[7] 検索時間
[8] ソート時間

This document is provided by JAXA.

7.7　ジョインの特性比較

使用したデータと測定結果：

ist_tdz_16 と TD_XY_16 を使用した。

NNI では値の種類数が大きいとき、SVL の共通化処理のコストが高くなる。UNION よりもさらにパフォーマンスが悪いのは ist_tdz_16 に加えて td_xy_16 の SVL のサイズが大きいためである。

表 5. NNI と SQLite のジョイン時間比較

処理 (単位 ms)	NNI	SQLite 初回	SQLite 2 回目
create table join_16 as select * from ist_tdz_16 inner join td_xy_16 on ist_tdz 16.datetime = td_xy_16.datetime;	9,580	170,631	36,205

7.8　集計の特性比較

使用したデータと測定結果：

ist_tdz_16 を使用した。

第 1 の測定は次元を指定しない集計である。第 2 の測定は tdz を次元とする集計である。NNI の実装では、次元の値の種類が小さい場合、キューブ集計を行う。キューブ集計はソートを用いた集計よりも高速で、今回の測定での高速性に繋がった。

表 6. NNI と SQLite の集計時間比較

処理 (単位 ms)	NNI	SQLite 初回	SQLite 2 回目
select count(ist),max(ist),min(ist),avg(ist),sum(ist) from ist_tdz_16;	274	32,071	20,645
select tdz,count(ist),max(ist),min(ist),avg(ist),sum(ist) from ist_tdz_16 group by tdz	449	64,974	34,104

7.9　更新の特性比較

使用したデータと測定結果：

ist_tdz_16 を使用した。以下のストーリーで一連の更新処理を構成した。

1. ist_tdz_16 からカラム tdz について、tdz＝0 のレコードを検索し（4,063,339 レコードが Hit）、これを別テーブルに export する。
2. ist_tdz_16 の上記の 4,063,339 レコードについて、カラム ist の値を -1 にする。
3. ist_tdz_16 の上記の 4,063,339 レコードを削除し、95,551,577 レコードにする。
4. ist_tdz_16 に上記 export したテーブルを UNION し、99,614,916 レコードに戻す。

This document is provided by JAXA.

上記 4 の UNION の処理時間が 7.6 より小さいのは ist_tdz_16 の SVL が UNION 処理で変化せず、ist_tdz_16 側の SVL の共通化処理が簡素になったためだと思われる。

表 7. NNI と SQLite の更新時間比較

処理 (単位 ms)	NNI	SQLite
create table ist_tdz_tdz_0_16 as select * from ist_tdz_16 where tdz=0; .save /home/ubuntu/note/input/apollo/ist_tdz_tdz_0_16.db	115[9] +341[10]	2,690
update ist_tdz_16 set ist=-1 where tdz=0;	60	9,586[11]
delete from ist_tdz_16 where tdz=0;	622	17,367[12]
insert into ist_tdz_16 select * from ist_tdz_tdz_0_16	5,092	5,716

8 NNI の関連技術と研究

NNI はカラム指向のインデックステクノロジーであり、データベースではない。しかし、同様にカラム指向ではあるデータベースと比較するのは有意義である。これらのデータベースでは SVL を導入している例も多い。

Sybase-IQ [6] はインメモリではなくディスクベースのシステムである。ランダムアクセスできないので性能低下を補うために様々なインデックス(9 種類)が搭載されている。またカラムの中間に新データを挿入するのに時間がかかるため、新データの中間挿入は禁止し、末尾への追加のみに制限している。

C-Store [8]、Vertica [9] もディスクベースである。大規模データ対応のためシェアドナッシング方式の超並列方式を導入しているが、このため機能的な制約が大きく、またシングルタスクとなりジョブがキューイングされるため多数のユーザの同時使用はしにくい。また、上記と同じく新データは末尾に追加のみに制限している。

SAP-HANA [10) 11] はインメモリであり、NNI の特許群のライセンス先の一つである。原則的としてインデックスは使わない。インメモリを活かして新データの中間挿入ができるが、1 つのカラムを多数の小ブロックに分割して格納することで、中間挿入の速度を上げて OLTP に対応している。しかしこのために、ソート他のデータベースの大域を扱う処理での速度低下が激しい。

なお、呼称が類似している「分散ソート済みカラム指向データベース」と呼ばれるものがあり、その代表例として BigQuery [7] がある。BigQuery でソート済みなのはノードに割り付けられたリージョンとリージョン内のレコードであり、項目毎にソートされているわけではない。分散構造であるため、機能上の制限が多く、例えば新データの中間挿入は禁止され、末尾への追加のみになっている。

---

[9] 検索時間
[10] 検索の結果できたサブテーブルを別のテーブルとして独立させる時間
[11] 一度実行済みに関わらず、SQL では再度実行が必要な ”tdz=0” の推定実行時間 1,320 ms を含む
[12] 上記に同じ

This document is provided by JAXA.

9　NNI の拡張

NNI は拡張の途上にある。当初 1 つだった NNI は、現時点で 3 つの形態に拡張されている。本論文では単一コンピュータ上のインメモリで表形式データの編集・加工・分析を行う、第 1 の形態の NNI を紹介した。この形態の NNI は今では Zap-In(Zap In-Memory)と呼ばれる。

9.1　単一コンピュータ上で、表形式データの編集・加工・分析を行う Zap-In

第 1 の形態の NNI である Zap-In は理論面ではすでにまとまり、ソフトウェアも存在し、本論文に記載したようにきちんとしたベンチマークを行うことができる。JAXA には株式会社セックの XML データベースである Karearea に組み込まれ、2003 年ころから SODA システム、人工衛星観測システムに導入されたことがある [3)]。航空宇宙以外にも原子力、流通、金融、製造、情報通信、マーケティング、医療、教育などの分野で活躍している。その特許 [12-19)] のライセンス先は NEC、富士通 BSC、SAP（独）などである。しかし、長い年月を経ても幅広く使われているとは言えない。この閉塞状況の中、本論文で Zap-In の体系を示し、ベンチマークでも有用性を確認し、改めて世に問うことは大変意義があったと考えている。

9.2　クラウド間で表形式ビッグデータを組み合わせ、検索し、ダウンロードする Zap-Over [22, 23)]

第 2 の形態の NNI は Zap-Over(Zap Over the Internet)と呼ばれる。Zap-Over は単一コンピュータ上のデータのみを扱うわけでもなく、インメモリでもない。しかし、表形式データの成分分解を定義し、その上で動くアルゴリズム群で構成される点は変わらない。Zap-In の適用限界が 20 億レコードであるのに対し、Zap-Over は数兆レコードを扱えるように設計されている。

Zap-Over は世界各地のクラウドに散在する様々なビッグデータを、データ利用者が選び、所望の形に組合せてダウンロードすることを可能にする。Zap-Over は D5A というビッグデータファイルフォーマットを基盤とする。データ提供側はビッグデータを D5A ファイルにしてファイルサーバに置くだけでよい。データ利用者はクラウド間で必要な D5A ファイルを選択して組合せ、検索して、必要な部分をダウンロードする。時系列データなど共通キーがあればさらに柔軟な出力レイアウトを指定できる。この D5A ファイルを利用すると、ビッグデータの提供サービスの著しいコストダウンが計れる上、データ利用者がデータ提供者に縛られることなく、セルフサービスで多様なビッグデータをクラウド群から得ることが可能になる。

Zap-Over は 2013 年に一旦、東京国税局に採用されたがその後もブラッシュアップが続いている。2018 年、DEIM2018[13] で発表する機会を得た。現在は JAXA との共同研究の形で進めており、2020 年 2 月、その中間報告を宇宙科学情報解析シンポジウムで発表する機会を得た。Zap-In の真価は様々なビッグデータが入手可能になったときに発揮されることから Zap-In にとって Zap-Over との連携は最も注力すべき開発課題となっている。別の機会で論文形式での Zap-Over の発表を検討中である。

[13] データ工学と情報マネジメントに関するフォーラム　https://db-event.jpn.org/deim2018/

This document is provided by JAXA.

9.3 Zap-In を超並列環境向けに拡張した Zap-Mass [24)]

第 3 の形態の NNI は Zap-Mass(Zap Massive Parallel)と呼ばれる。Zap-In は優れた技術であるが、適用限界が 20 億レコードと大きいとは言えない。それ以上にデータ量を拡大するとメモリが足りなくなる他、CPU とメモリのバランスが悪化し、効率が低下する。この課題を Zap-Mass は克服する。Zap-Mass は多数のコンピュータ上でインメモリで Zap-In と同様の処理を行う。

そのアルゴリズムは各コンピュータ上で Zap-In でローカル処理を行った後、コンピュータ間でグローバル処理を行うことを基本とする。コンピュータ間で行われるグローバル処理は行列式の積和演算のように順序を変えても成立する。この性質を生かすと表形式データの適切な格納区画をコントロールでき、また多人数で共有するシステムとするために必要なプリエンプティブなマルチタスクが容易になる。

将来、NNI はこの仕組みを使った DB プロセッサ上で動かすのが当たり前になるかもしれないと考え、開発を継続している。

9.4 表形式データ以外への拡張 [3)]

Zap-In/Over/Mass には①表形式のビッグデータを扱う、②成分分解を行ってデータそのものに内在する順序関係を利用したアルゴリズムで高速化する、という 2 つの共通項がある。うち、「表形式」というのは理論的には必然性がない。事実、成分分解を行って XML データベースのエンジンを実現する試みがあり、理論的には成功した。その際は、幅優先・深さ優先の 2 つの標準形を定義して OrdSet を拡張した。複数の特許を取得できた。しかし、その拡張した OrdSet の有意義な使い方を提案できなかった。現在、表形式以外への NNI の拡張はフリーズされている。

10 まとめ

宇宙科学分野では観測データなど膨大なデータが存在し、それらのビッグデータを組み合わせて解析するニーズが日常的に存在する。その多くはインデックス設計などのチューニングにいちいち時間をかけられない非定型処理である。またそれらの処理時間は長大になりがちで大幅に短縮する必要がある。

そこで著者らはこのようなニーズに応えるために表形式データを幅広いケースで高速処理できる自然数インデックス（NNI: Natural Number Index）の利用を提案した。既存のインデックスは処理対象データの外部にあるデータ構造を利用し、それを本質的には一つのアルゴリズムでアクセスする単用途である。NNI は処理対象データに内在する順序関係を利用し、多くのアルゴリズム群でアクセスする多用途のインデックスである。

その NNI の仕組みを述べる。全ての表形式データは NNI が定める成分に一意に成分分解できる。するとそれらの成分を介して、表形式データに内在する順序関係を使うアルゴリズム群が設計可能になる。NNI の実体はそのアルゴリズム群である。

① NNI が定める成分分解の下では、表形式データに内在する順序関係は、すべてのカラム、すべてのカラムの取り合わせ、すべての部分集合に等しく存在する。そのためその順序関係を使う NNI のアルゴリズム群は任意のカラム、任意のカラムの取り合わせ、任意の部分集合に対する処理を高速化できる。

This document is provided by JAXA.

② またこの順序関係は、検索・集計・ソート・計算・データ変換および関係代数演算の基盤である。それを利用する NNI のアルゴリズム群は検索・集計・ソート・計算・データ変換およびビッグデータを効率的に処理する上で重要なジョインなどの関係代数演算を幅広くカバーする。
③ その NNI の検索・ソート・集計などの個々の処理は既存のインデックスより桁違いに高速であることが多い。それは本論文中のベンチマークで示された。
④ さらにこの順序関係はデータの更新に伴い自動的に更新され、新たに生成された表形式データ（例えば集計結果テーブル）にも最初から存在する。従って NNI のアルゴリズム群は新たにインデックス用のデータ構造を作成する手間なく、更新・生成直後の表形式データを直ちに高速に処理できる。
⑤ 加えて NNI のアルゴリズム群は処理の結果を成分のまま出力する。そのため NNI は単独処理を効率的にカスケードして複合処理を作ることができる。

つまり、NNI は表形式データの①全域をカバーでき、②多様な処理を行え、③個々の処理で高速で、④更新・生成後で直ちに処理でき、⑤個々の処理を効率的に結合して複合処理を作ることができる。これらは外部構造を作らずにデータそのものに内在する順序関係を使う NNI ならではの特長である。

NNI の個々の処理の性能について考察する。NNI は検索・ソート・集計など個々の処理についても既存のリレーショナルデータベースシステムより桁違いに高速であることが多い。アポロ 11 号と 16 号の観測データを使ったベンチマークでは事前のインデックス設計が不要な自然数インデックスを使ったシステムと、事前にインデックスを設定した SQLite の実行速度を比較しても、多くのケースで自然数インデックスが遙かに高速であった。

このベンチマークは比較対象の SQLite の実用領域に合わせる必要があったため、自然数インデックスの実用領域である 10 億レコードの 1／10 である 1 億レコードで行われた。1 億レコードより大きいデータにおいて自然数インデックスのアドバンテージはさらに拡大する筈である。また、自然数インデックスが最も効果を発揮する仮想ジョインテーブルの処理では NNI の効果はさらに大きい筈である。

一方、自然数インデックスで少数のレコードを検索する性能は SQLite に比べて低かった。しかし実際にかかっている時間は 150 ミリ秒程度であり、自然数インデックスが使われるデータ解析などの用途で支障を来すことは考えにくい（7.4 参照）。

このように宇宙観測分野の研究で求められるデータベースに対する要求と自然数インデックスの特性は良く整合する。そのため自然数インデックスをうまく活用すれば、宇宙観測分野の研究が加速されることが見込める。また、その自然数インデックスを導入して対話型でビッグデータ処理を実行する実用システムは既に実現されている。あとは宇宙科学分野の情報処理の一般的な作業環境にフィッティングするだけで実用化できる筈である。

つまり NNI を使えば宇宙科学分野のさまざまなビッグデータ処理を設計時と実行時の両方で大幅に効率化できる。その結果、NNI は日々のビッグデータの非定型処理を容易にし、加えてこれまでやれなかった処理も実行可能にする。NNI のソフトウェアの開発もフィッティングを残すだけのところに来ている。NNI が活用され宇宙科学分野の様々な研究を加速する日は近いと期待している。

This document is provided by JAXA.

参考文献

[1] 増永良文, “リレーショナルデータベース入門—データモデル・SQL・管理システム [第 3 版]”, Information & Computing-116, サイエンス社, 2003, P.220.

[2] 北川博之, “データベースシステム(改訂 2 版)”, オーム社, 2020, P.155.

[3] 古庄晋二, “汎用超高速データベース処理技術: 多様なデータ構造と超並列処理への普遍的アプローチ- RealTimeVIDP”, 東大総研 2005.

[4] Donald E. Knuth, “Art of Computer Programming, Volume 3: Sorting and Searching”, Addison-Wesley, 1973.

[5] 製品情報 ESPERiC https://www.ess-g.com/esperic

[6] ホワイトペーパー, “クラウドにも対応！DWH に最適化された最高の ROI を実現するデータベース Sybase IQ” https://wp.techtarget.itmedia.co.jp/contents/12605

[7] “Google BigQuery のドキュメント” https://cloud.google.com/bigquery/docs/?hl=ja

[8] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, Stan Zdonik, “C-Store: A Column-oriented DBMS”, Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005. http://db.csail.mit.edu/projects/cstore/vldb.pdf

[9] “Vertica とは”, Vertica 技術サイト, 2015. http://vertica-tech.ashisuto.co.jp/about-vertica/

[10] “[図説]インメモリーコンピューティング SAP HANA®のテクノロジー解説: 真のリアルタイム経営を支えるリアルタイムデータプラットフォームの実力とは”, 2015. https://www.intel.co.jp/content/dam/www/public/ijkk/jp/ja/documents/white-papers/xeon-e7-in-memory-computing-sap-hana-technology-paper.pdf

[11] 三原健一, “HANA はどうやって行を識別しているのか”, Oracle 技術者から見た、SAP HANA, 2017. https://enterprisezine.jp/dbonline/detail/10198

[12] NNI の検索・集計・ソートの特許 https://patents.google.com/patent/US6643644

[13] NNI の JOIN 作成方法（1） https://patents.google.com/patent/US6721751B1

[14] NNI の更新・トランザクション https://patents.google.com/patent/US6973467B1

[15] CSV などの表形式データを高速に成分分解する方法 https://patents.google.com/patent/US7225198B2

[16] NNI の JOIN 作成方法（2） https://patents.google.com/patent/US7184996B2

[17] NNI の JOIN テーブルのチェーンを高速にツリー形式に変換する方法 https://patents.google.com/patent/US7467130B2

[18] NNI の微小部分への処理効率を高める方法 https://patents.google.com/patent/US7882114B2

[19] NNI でパラレルソートを実現する方法 https://patents.google.com/patent/US8065337B2

[20] アポロ月地震データ公開システムの開発 https://ci.nii.ac.jp/naid/110009140271

[21] NNI のデモムービー（技術名を Zap-In と記載している） https://www.youtube.com/watch?v=0wtICPf4WJc&feature=youtu.be

[22] Zap-Over の基本となる、所望部分のみのソート結果を得る方法 http://turbodata.sakura.ne.jp/Zap-Over%20Pat%20Application.pdf

This document is provided by JAXA.

[23] Zap-Over のデモムービー　https://www.youtube.com/watch?v=2Pg9tVocb9M&feature=youtu.be
[24] Zap-Mass の基本となるグローバル処理の実現方法
http://turbodata.sakura.ne.jp/Zap-Mass%20Eng%20Pat.pdf

This document is provided by JAXA.

付録：自然数インデックス上のアルゴリズム詳細

付録 1　実テーブル上の検索のアルゴリズム

図 A-1 の範囲検索、図 A-2 のリスト検索を例に説明する。

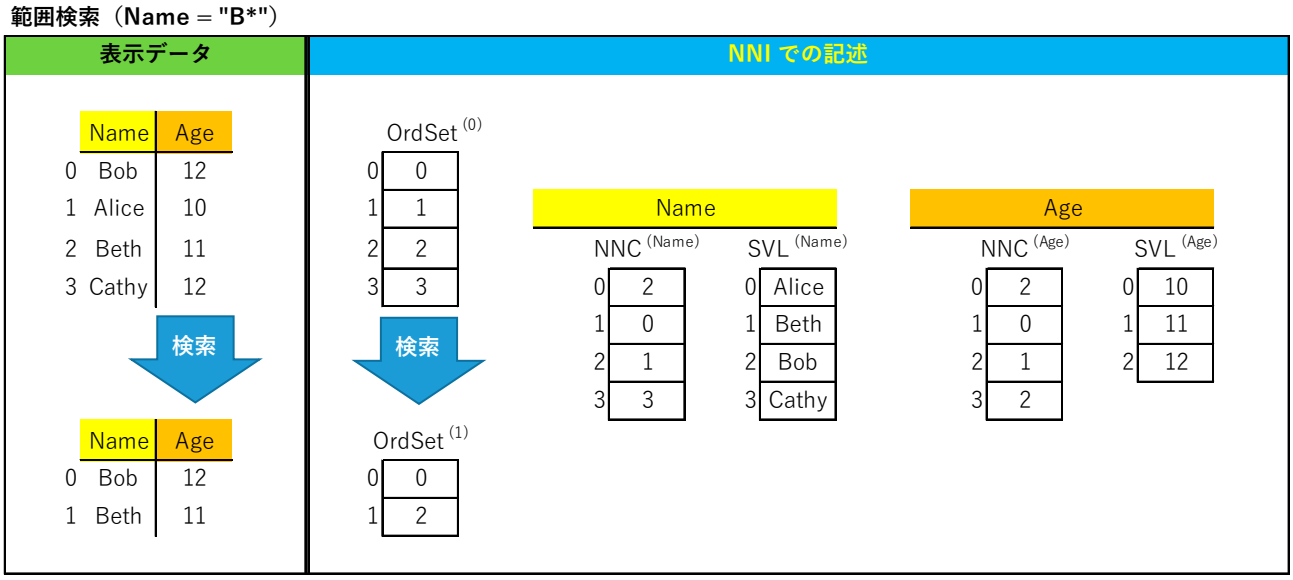


図 A-1. 範囲検索

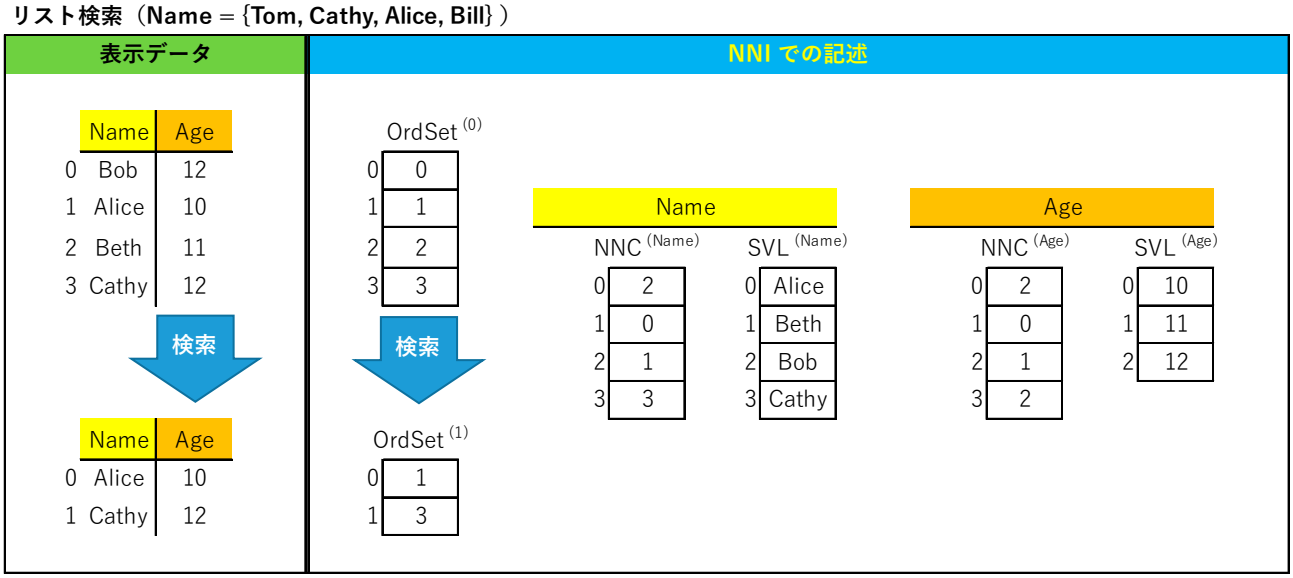
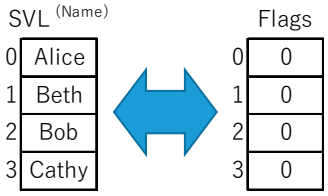


図 A-2. リスト検索

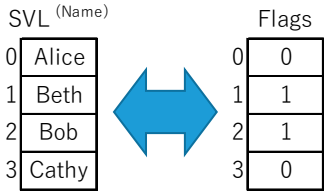
This document is provided by JAXA.

検索対象のSVLと同じサイズのフラグ配列を用意し、初期化する。

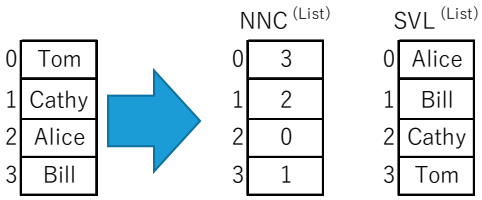


ケース1.　範囲検索（Name = "B*"）の時

条件に当てはまるSVL上の開始位置、終了位置を求め、
その区間のFlagsをマークする。
（オプティマイズ1.　区間の先頭と最後を特定するだけで中間は適否を判断する必要は無い）
（オプティマイズ2.　区間の先頭と最後のアドレスのみを保持し、フラグ配列を使わない選択もある。）



ケース2.　リスト検索（Name = {Tom, Cathy, Alice, Bill}）の時



与えられたリストを成分分解する。
SVL (List) を得る。

SVL (List) と SVL (Name) を先頭から順次比較し、両方に存在する値があれば、
その値の SVL (Name) 中の位置のフラグ配列の要素をマークする。
図の例では、**(Alice, Alice)**, (Bill, Beth), (Bill, Bob), (Cathy, Bob), **(Cathy, Cathy)** と比較し、
AliceとCathyの位置のフラグをマークする。
（オプティマイズ3.　SVL (List) のサイズ << SVL (Name) のサイズ　であるので、
高速化のためにSVL (Name) の要素をジャンプしながら比べる方法もある。）

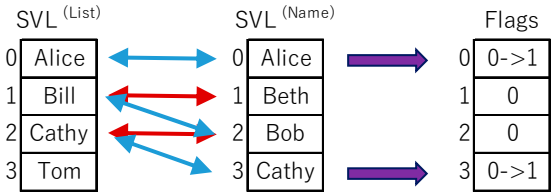


図 A-3.　フラグ配列のマーク

This document is provided by JAXA.

検索は、二つのステップで構成される。

1. フラグ配列のマーク(図 A-3)

   SVL と同じサイズのフラグ配列を用意し、SVL の各値が検索条件にマッチすれば、その値の格納位置に相当するフラグ配列の要素をマークする。

   範囲検索の場合は、フラグ配列のマーク領域の先頭と末尾を求め、その区間をすべてマークすれば良く、ほぼ O(log(n)) でできる。

   リスト検索の場合は、与えられたリストを成分分解し、作成された SVL と、検索対象カラムの SVL を見比べてマークを行う。

2. ヒットテスト(図 A-4)

   Flags[NNC $^{(Name)}$ [OrdSet $^{(i)}$ [j]]] を評価し、マークされていたら、OrdSet $^{(i+1)}$ に OrdSet $^{(i)}$ [j] を格納する。ヒットテストが終われば、検索結果を表す、OrdSet が完成する。

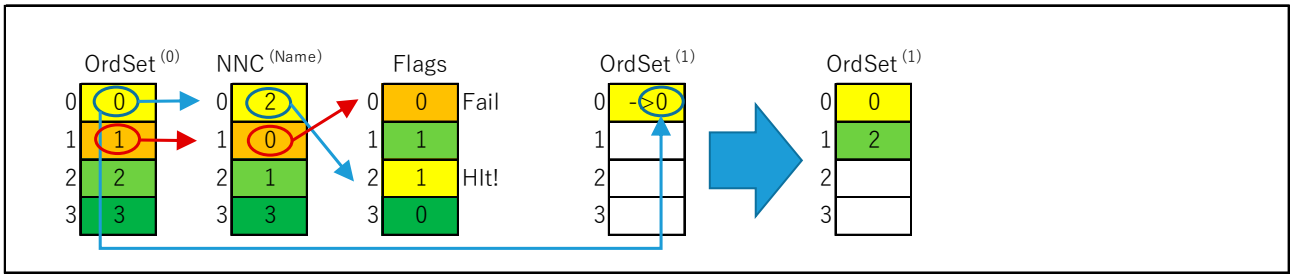


図 A-4. ヒットテスト

This document is provided by JAXA.

付録 2　実テーブル上のソートのアルゴリズム

図 A-5 のように Age でソートする例で説明する。

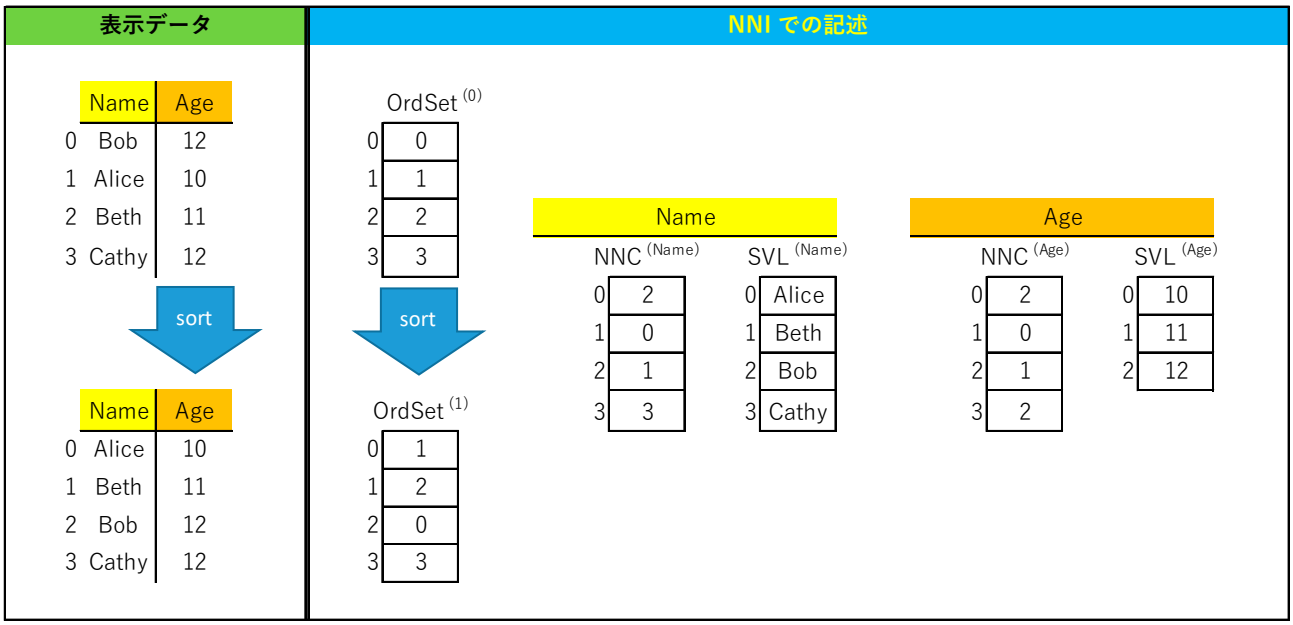


図 A-5.　ソートの概要

A2.1　ソートのステップ

以下の 3 つのステップで行う。

第1.　カウントアップ　（どの値が何回出現したかを数える）

第2.　累計数化　（各レコードの格納先のアドレスを算出する）

第3.　OrdSet の転送

A2.2　カウントアップのステップ(図 A-6)

各 SVL の要素が何回出現しているかを数える。

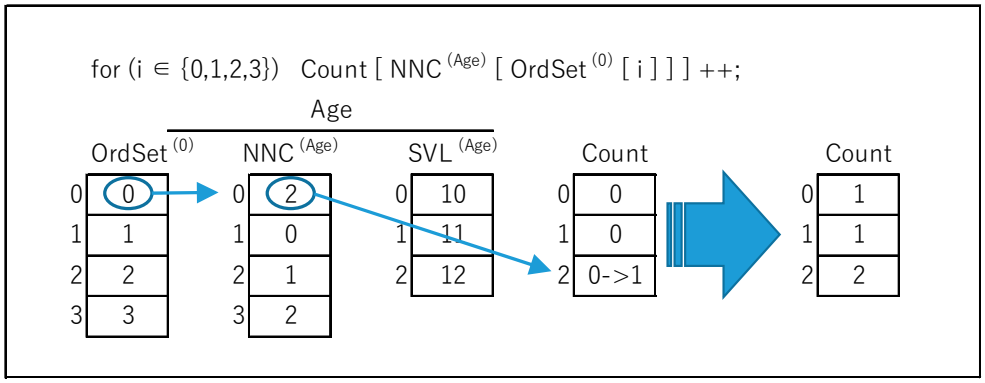


図 A-6.　各値が何回出現したかを数える

This document is provided by JAXA.

A2.3 累計数化(図 A-7)

Count を累計数化する。その際、累計数の格納位置は 1 つ後ろに下げる。こうすることで、Aggr は各値のソート結果となる OrdSet 上の格納開始位置を保持する。

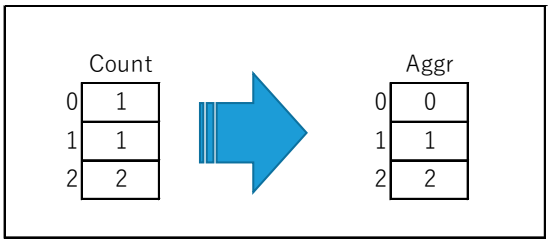


図 A-7. 累計数化

A2.4 転送(図 A-8)

下記の処理を行う。転送が完了すれば、OrdSet$^{(1)}$ が完成し、ソートが完了する。(なお、Aggr 配列は各値以下の出現数を表し、他の処理にも流用できる。例えば、仮想ジョインテーブルを構成する SAcm 成分は Aggr をそのまま使う。)

$$for\ (i \in \{0,1,2,3\})\quad OrdSet^{(1)}[\,Aggr[\,NNC^{(Age)}[\,OrdSet^{(0)}[\,i\,]\,]\,]\,{+}{+}\,] = OrdSet^{(0)}[\,i\,];$$

This document is provided by JAXA.

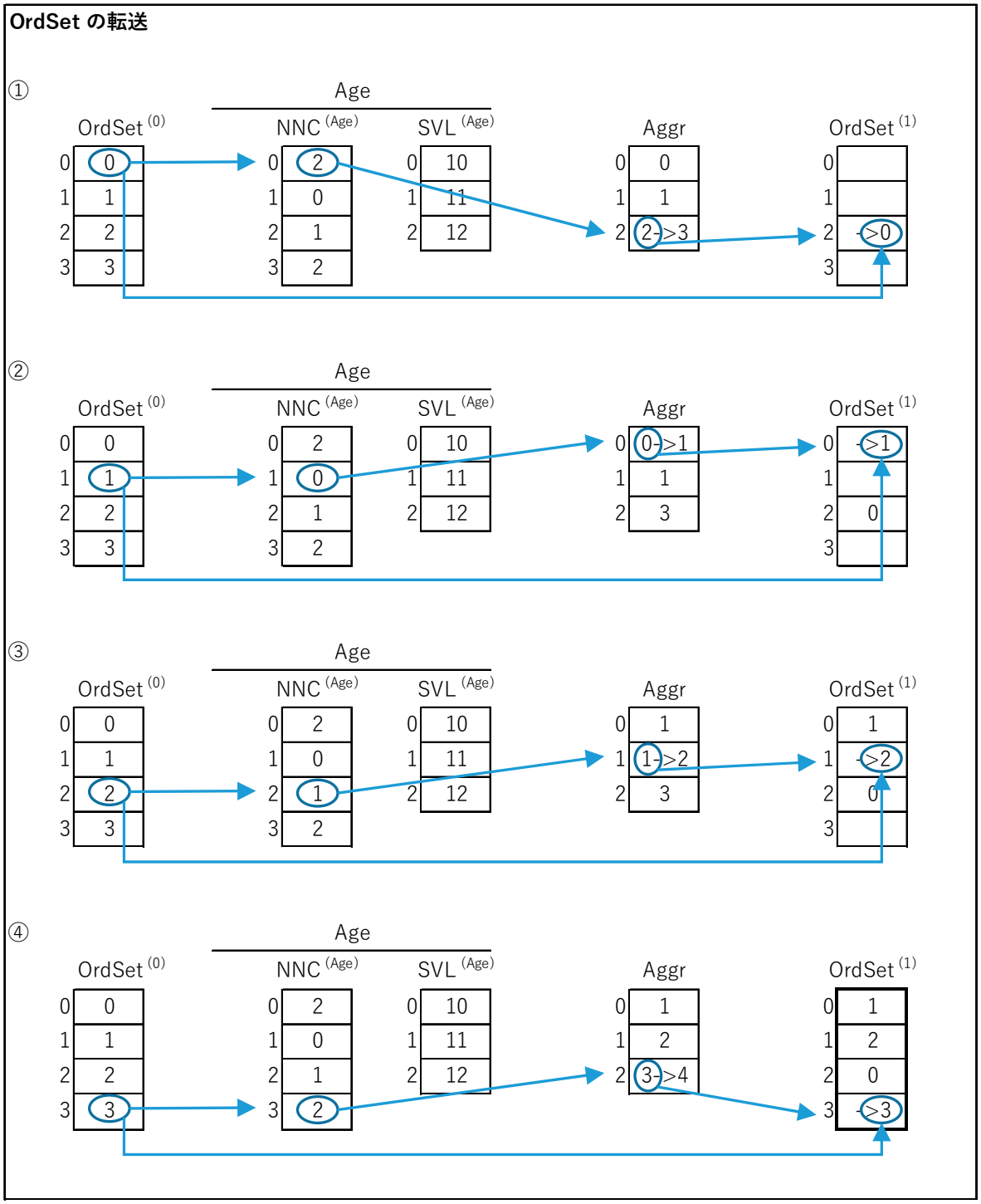


図 A-8.　転送

This document is provided by JAXA.

付録 3　実テーブル上の集計のアルゴリズム

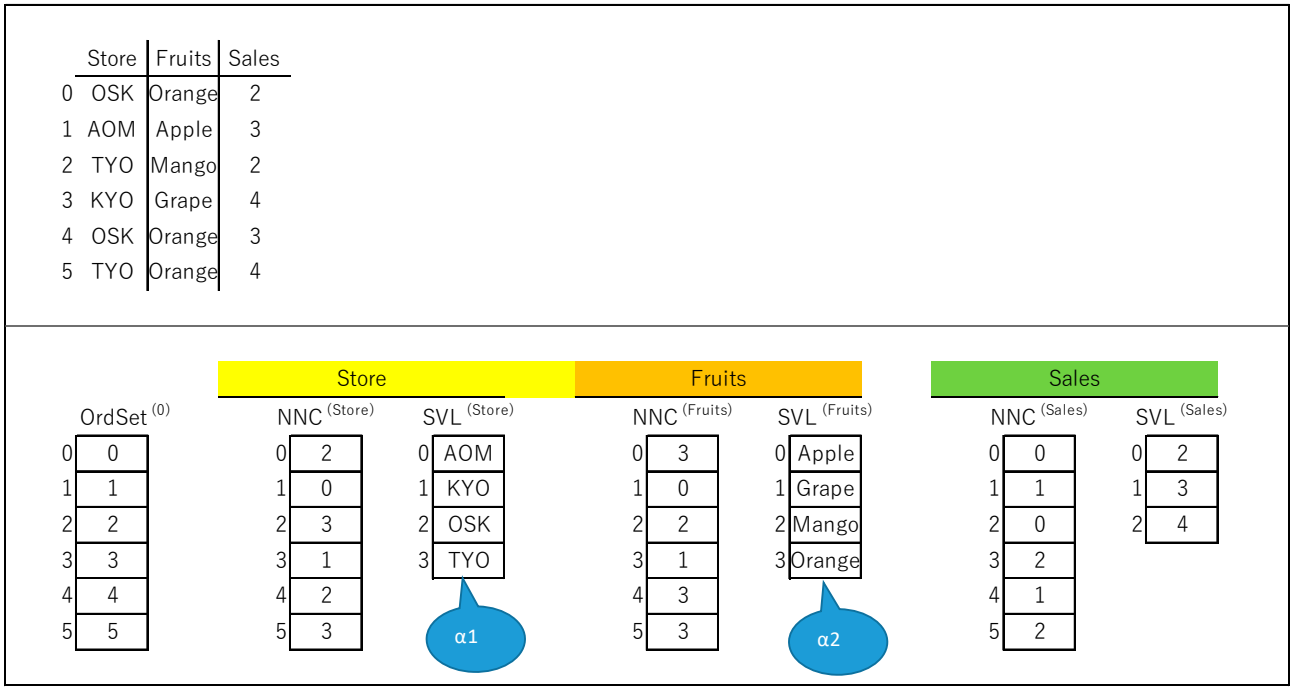
集計のアルゴリズムは大別して、①多次元キューブを使用する方法と、②ソートを使う方法がある。

① 多次元キューブ方式は実行できれば高速である。（しかし、多数の次元を使う場合、多次元キューブの体積が爆発して実行が困難になる）

② ソート方式はいつでも実行可能である。（集計結果テーブルのレコード数が最大でも集計元テーブルの OrdSet のサイズを超えないため）

ここでは、後者（②）のソート方式でアルゴリズムを説明する。

図 A-9 に集計対象テーブルを図示する。「どの店が何を何個売ったか？」という集計を例に考える。この場合、集計の次元は “Store”, “Fruits” で、測度は “Sum(Sales)” とする。



	Store	Fruits	Sales
0	OSK	Orange	2
1	AOM	Apple	3
2	TYO	Mango	2
3	KYO	Grape	4
4	OSK	Orange	3
5	TYO	Orange	4

	OrdSet (0)
0	0
1	1
2	2
3	3
4	4
5	5

	NNC (Store)	SVL (Store)
0	2	AOM
1	0	KYO
2	3	OSK
3	1	TYO
4	2	
5	3	

	NNC (Fruits)	SVL (Fruits)
0	3	Apple
1	0	Grape
2	2	Mango
3	1	Orange
4	3	
5	3	

	NNC (Sales)	SVL (Sales)
0	0	2
1	1	3
2	0	4
3	2	
4	1	
5	2	

図 A-9. 集計対象テーブル（上段：表示イメージ、下段：成分分解）

A3.1　集計アルゴリズムのステップ

集計は以下のステップで実現される。

1. 次元でソートする。
2. 次元の NNC が一致するものをグループ化し、グループ毎に測度を算出する。
3. 測度は成分分解されていない形式なのでそれを成分分解して集計を完成する。

A3.2　次元（“Store” / “Fruits”）でソートを行う(図 A-10)

（ソートの安定性を利用して、最初に “Fruits” でソートし、次に “Store” でソートを行う。）

This document is provided by JAXA.

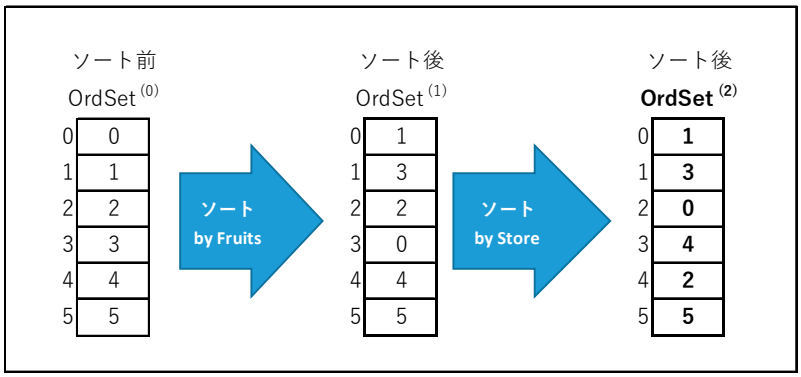


図 A-10.　次元でソートを行う

A3.3　次元の NNC が一致するものをグループ化し、グループ毎に測度を算出する(図 A-11)

図 A-11 に示すように OrdSet(2) を使うと次元の NNC 群の値を昇順に取り出すことができるので、次元値が同一なグループ毎に測度を算出できる。同一次元値の各グループの先頭の NNC の要素がそのまま集計結果テーブルの NNC の要素になる。

次元の SVL はそのままコピーして集計結果テーブルの次元の SVL にすることができる。

従って、この段階で集計結果テーブルの次元の NNIF が完成する。

This document is provided by JAXA.

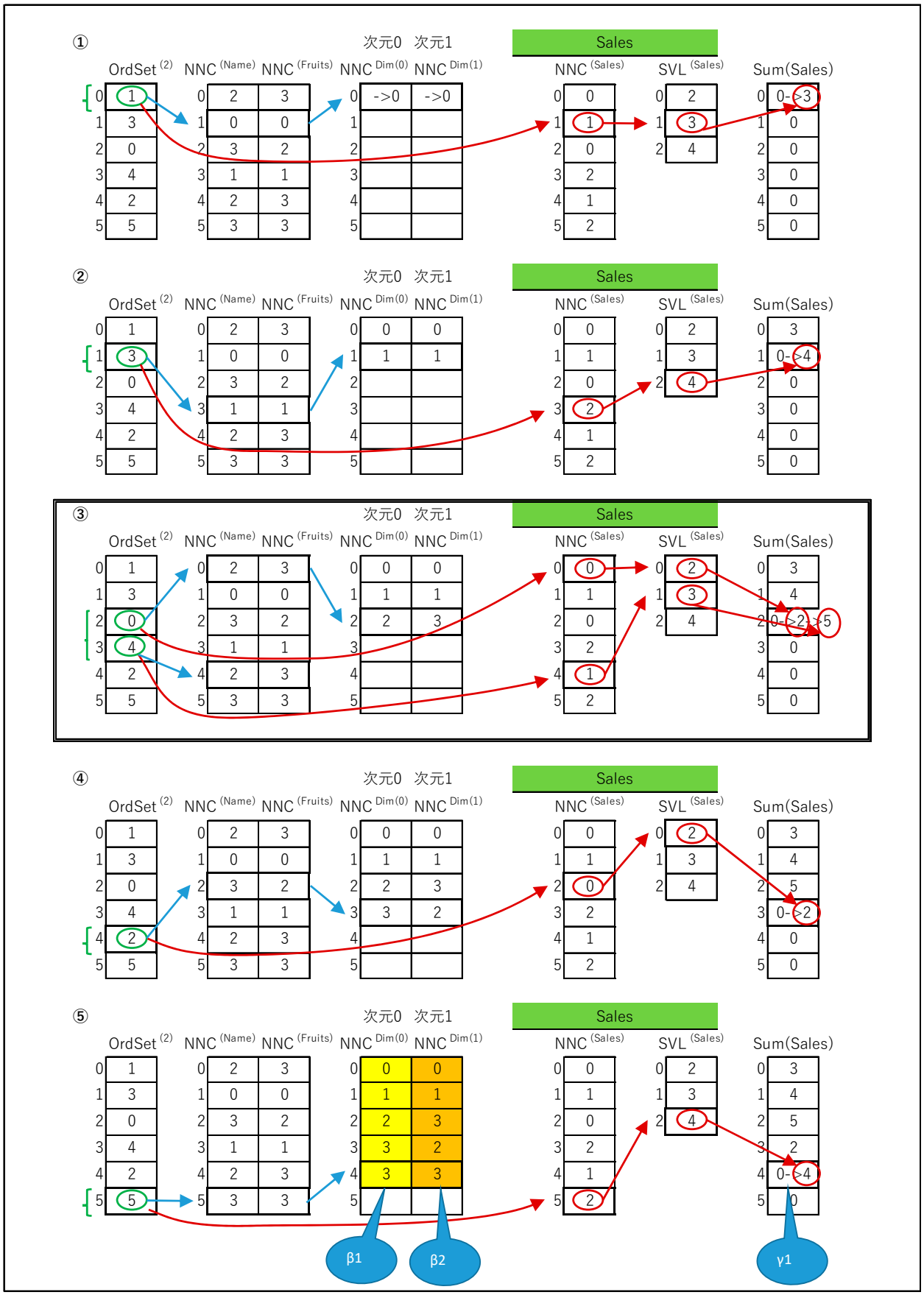


図 A-11. NNC が一致するものをグループ化して測度を算出する

This document is provided by JAXA.

A3.4　測度を成分分解する(図 A-12)

測度は成分分解されていない形式なのでそれを成分分解して集計を完成する。

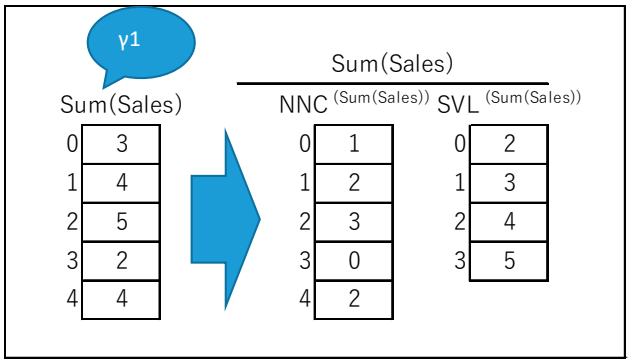


図 A-12．測度を成分に分解する

図 A-13 は完成した各成分を表示している。各成分の作成方法を以下に記す。多くの成分が効率的に作成されたことが分かる。

OrdSet (2)　：	0 からの連番をセットしたのみ。
Store と Fruits の NNC：	図 A-11 の β1/β2 をそのまま使う。
Store と Friuts の SVL：	集計の元テーブルの α1/α2 をそのままコピーしたもの。
測度（Sum(Sales)）：	図 A-12 の γ1 を成分分解したものを使う。

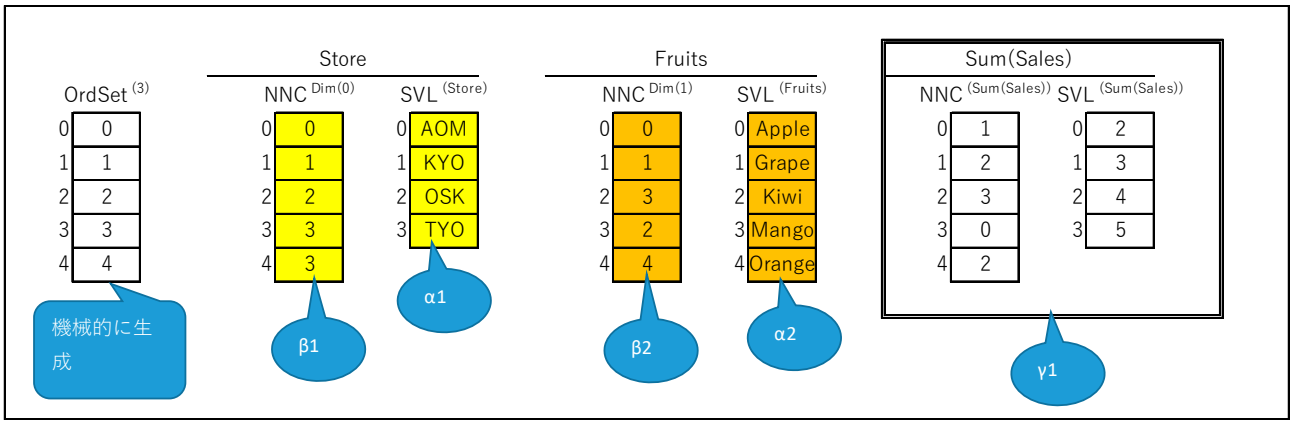


図 A-13．集計の完成

This document is provided by JAXA.

付録 4　SVL の共通化のアルゴリズム

図 A-14 に示す、テーブル A とテーブル B の Name カラムの SVL を共通化するアルゴリズムを説明する。SVL の共通化処理に伴い、2 つの NNC も更新される。この処理は UNION、ジョイン、データの上書きなどに伴って実行される。

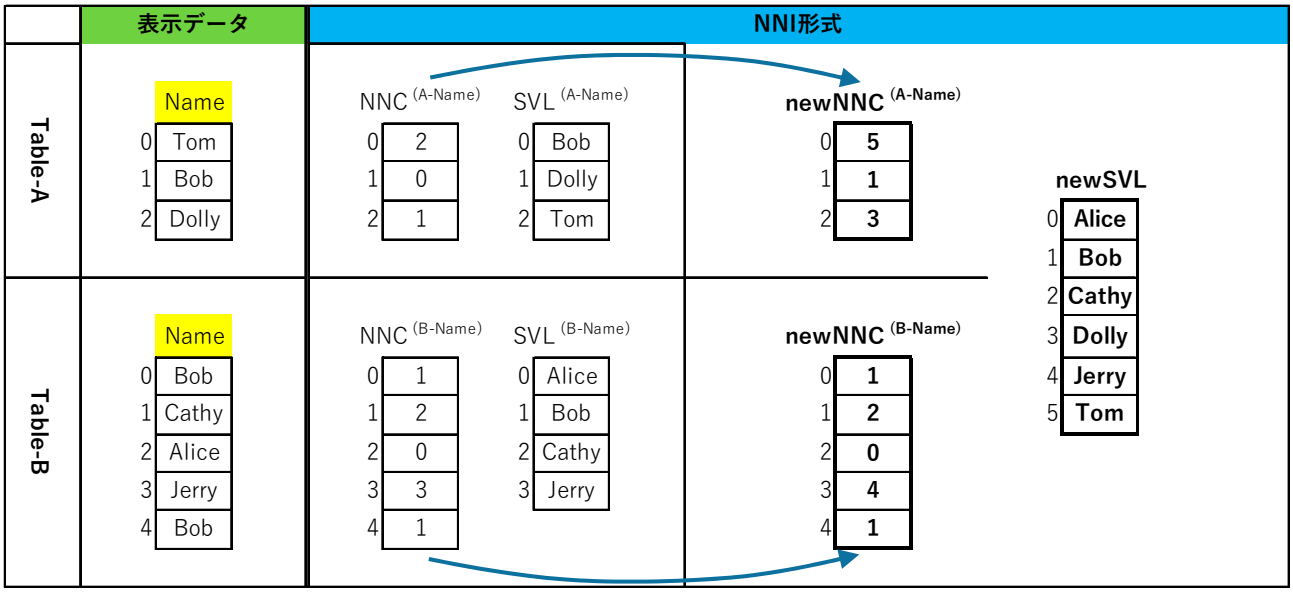


図 A-14. SVL の共通化処理

A4.1　newSVL と変換配列の作成(図 A-15)

図 A-15 に示すように、SVL (A-Name) および SVL (B-Name) のサイズに等しい、変換配列 Conv0/Conv1 を作る。同時に、二つの SVL の各値を格納する先となる、newSVL の格納領域を作成する。

次に、SVL (A-Name) および SVL (B-Name) の双方の値を先頭から比較する。図 A-15 中の各ステップ⓪～⑤について説明する。

①

- “Bob” と “Alice” を比較する。 (青)
- 小さい方(Alice) を newSVL に格納する。 (緑)
- Conv1 に newSVL への格納位置(=0) を格納する。 (紫)
- SVL (B-Name) の比較位置ポインターを 1 つ下げる。 (赤)

②

- “Bob” と “Bob” を比較する。 (青)
- 小さい方(Bob) を newSVL に格納する。 (緑)
- Conv0/Conv1 に newSVL への格納位置(=1) を格納する。 (紫)
- 両方の SVL の比較位置ポインターを 1 つ下げる。 (赤)

This document is provided by JAXA.

③ 省略

④ 省略

⑤

・“Tom” しか残っていない（SVL (B-Name) にはもう値が無い）。（青）

・小さい方(Tom) を newSVL に格納する。（緑）

・Conv0 に newSVL への格納位置(=5) を格納する。（紫）

・SVL (A-Name) の比較位置ポインターを 1 つ下げる。（赤）

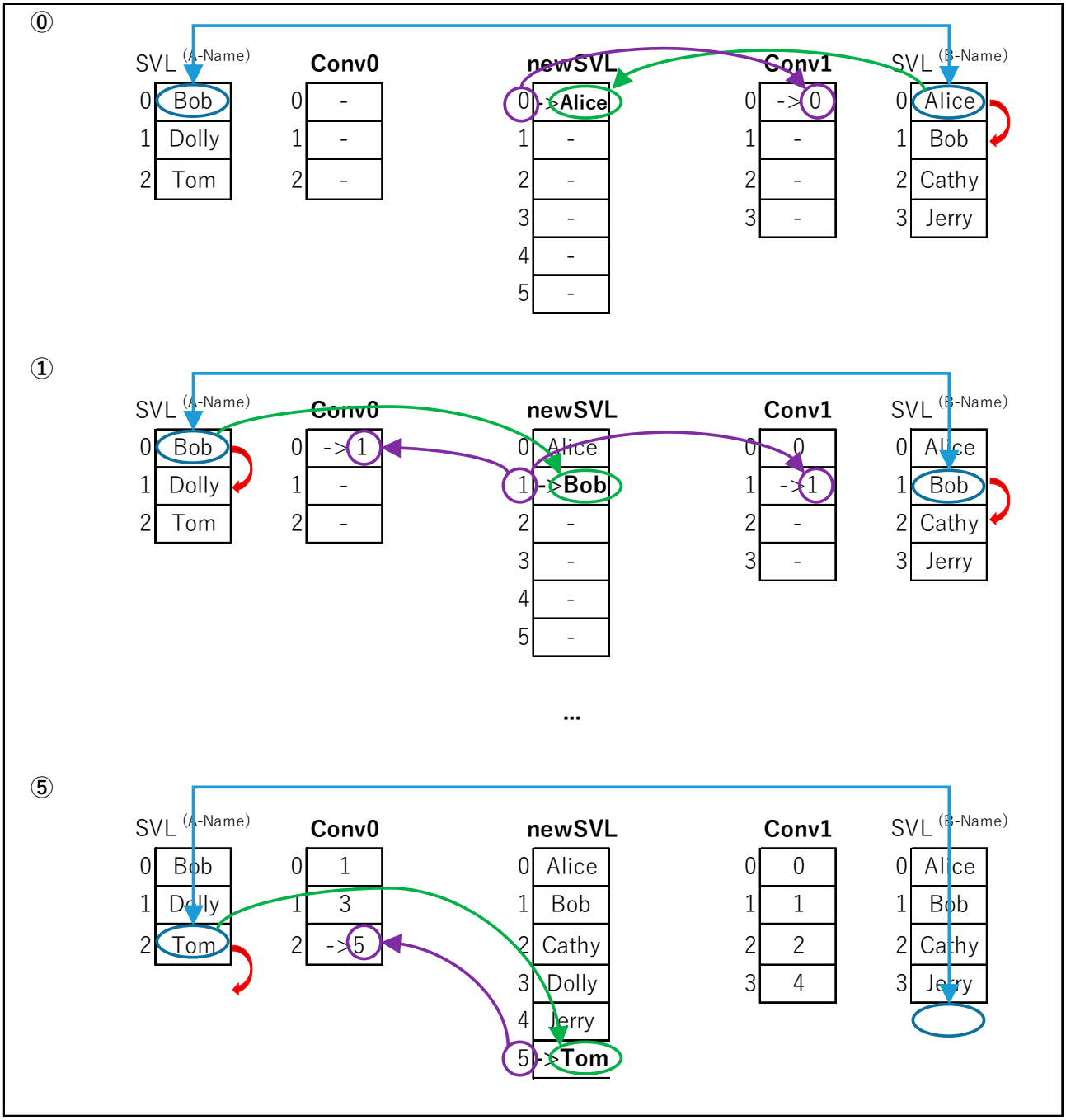


図 A-15. newSVL の作成

This document is provided by JAXA.

A4.2 NNC の変換(図 A-16)

図 A-16 に示すように、NNC (A-Name) と NNC (B-Name) をそれぞれ Conv0/Conv1 を参照しながら変換する。変換後の NNC (A-Name) と NNC (B-Name) を newNNC (A-Name) と newNNC (B-Name) と見なす。ここまでで、図 A-15 に示された newNNC./newSVL が完成した。

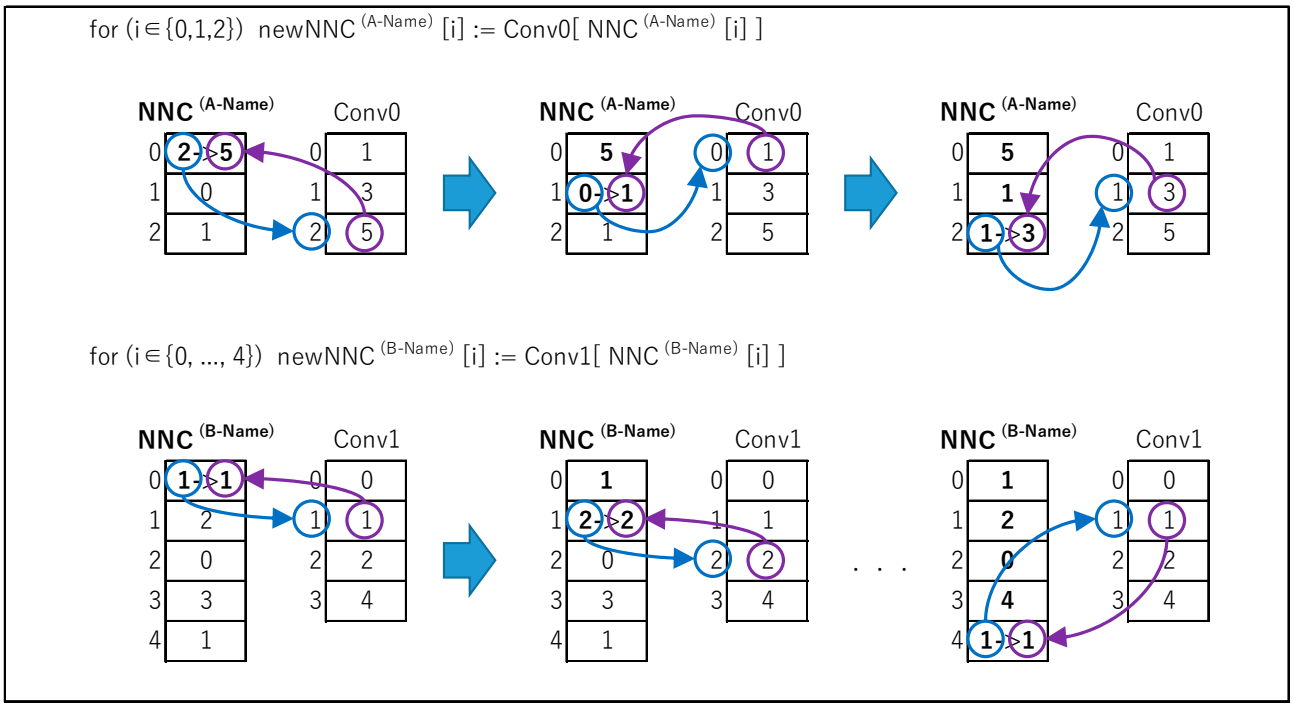


図 A-16. NNC の変換

This document is provided by JAXA.

付録 5　実テーブル上の UNION のアルゴリズム

図 A-14 に示されたテーブル A とテーブル B を UNION してみよう。最初に SVL の共通化を行い、newSVL、newNNC (A-Name) と newNNC (B-Name) を作成する。

次に、図 A-17 に示すように newNNC (A-Name) と newNNC (B-Name)を連結して newNNC (Name) を作成する。

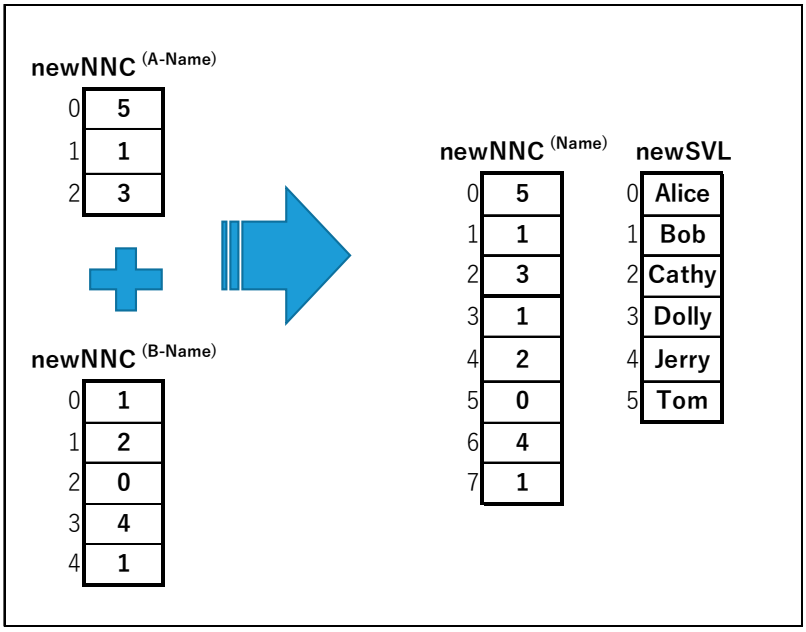


図 A-17．UNION の完成

This document is provided by JAXA.

付録 6　実テーブル上のデータの上書きのアルゴリズム

図 A-18 上段に示す上書き更新を行おう。まず更新データ（Data1）を成分分解する。

次に、更新データ（Data1）と、被更新データ（Data0）の SVL の共通化処理を行い、newSVL を完成する。（付録 4 参照）

最後に図 A-19 に示す、被更新データの NNC の上書き位置 (1, 3, 5) に、更新データの NNC の値を書き込めば上書きが完了する。

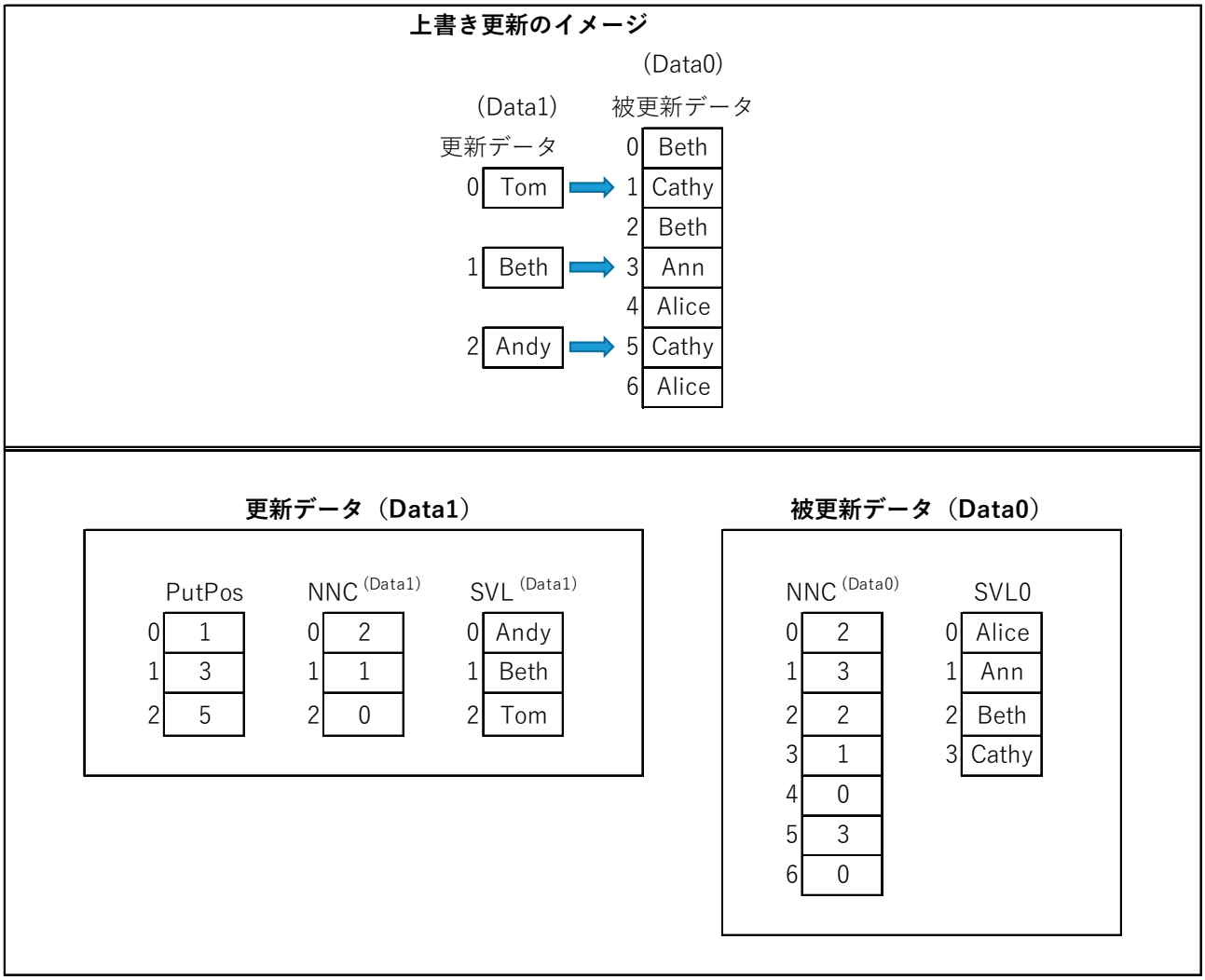


図 A-18. データの上書き更新

This document is provided by JAXA.

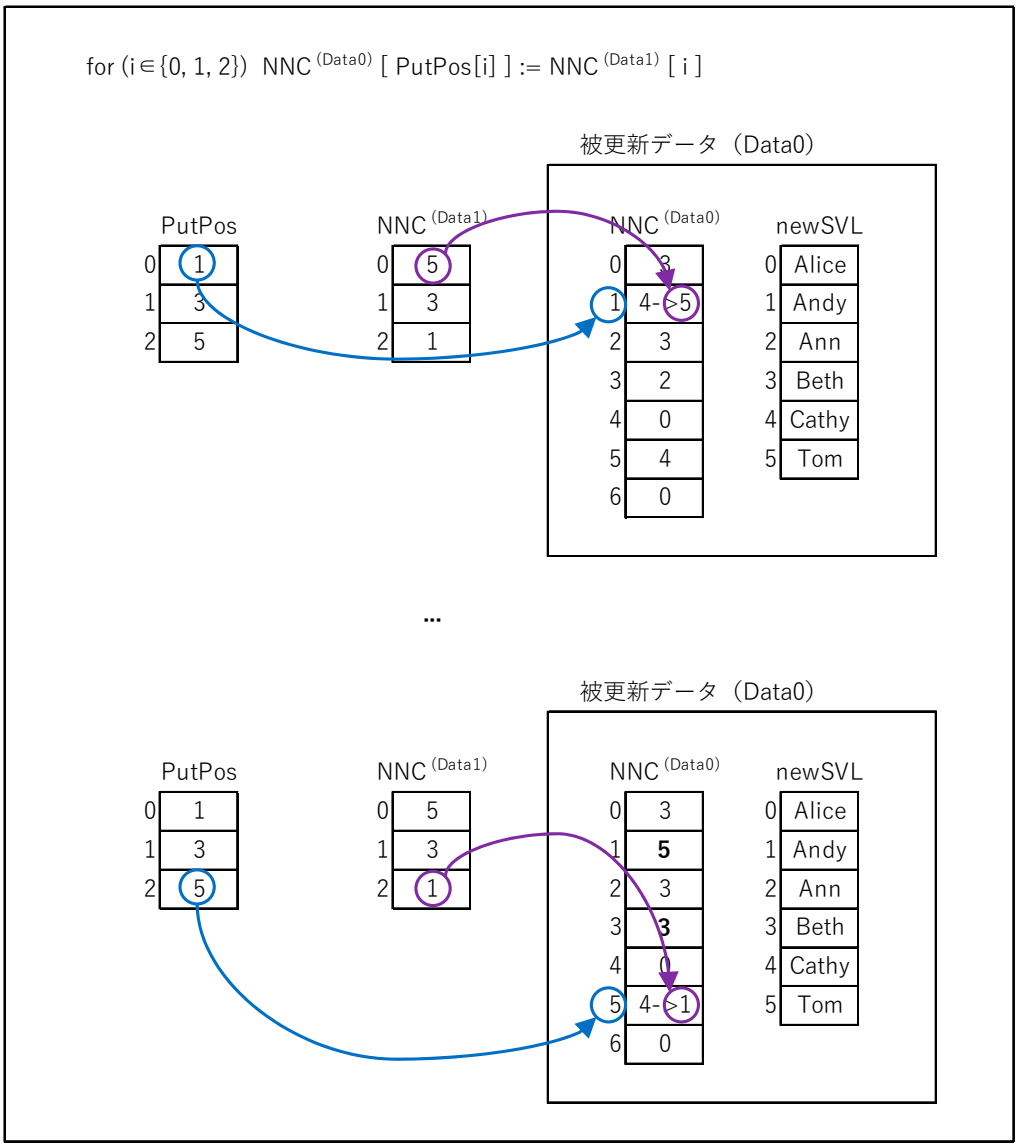


図 A-19．上書き位置（PutPos）に NNC (Data1) を配置する

This document is provided by JAXA.

付録 7　実テーブル上のレコード群の挿入のアルゴリズム

図 A-20 に示すテーブルにレコード群を挿入する操作を説明する。

挿入位置と挿入レコード数は図 A-21 に示す。

挿入に際して、null 値を定義する必要がある。整数では 2^{63}、浮動小数点では -INF、文字列では空文字列などを使う。空文字列と文字列の null 値の区別が必要な場合、別途工夫が要る。

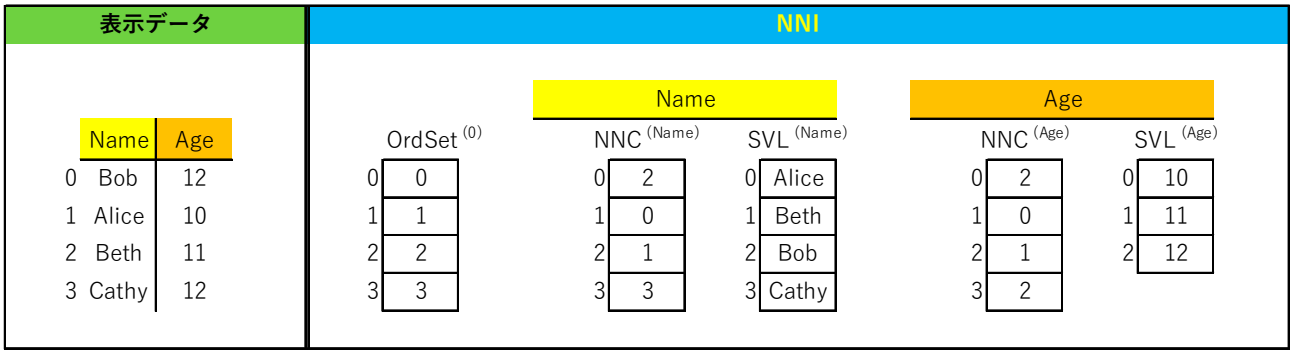


図 A-20.　レコード群の挿入を行う前のテーブル

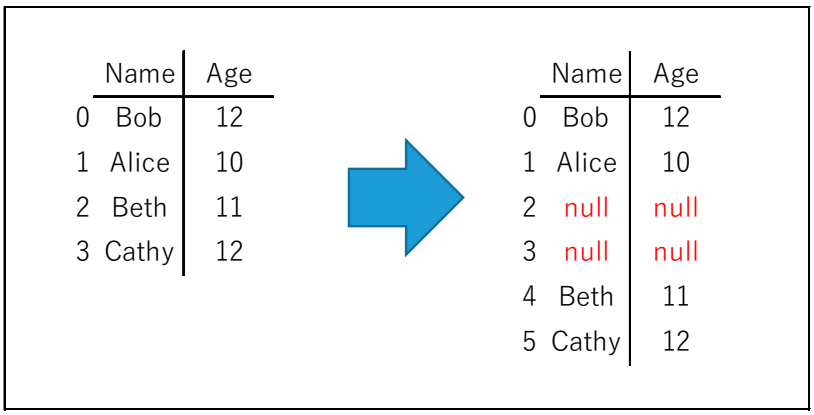


図 A-21.　例としてあげるレコード群の挿入

図 A-22 にレコード群の挿入により、各配列がどのように変化するかを示した。以下の操作を行う。

- ルート OrdSet は要素数が 2 つ増えただけで、0 から始まるユニーク・連番であることに変わりは無い。
- ルート OrdSet 以外の OrdSet は、挿入位置(=2)以上の値を持つ要素について、挿入行数(=2)だけインクリメントする。
- NNC は SVL に null が挿入されたことに伴い、既存の要素の値は 1 だけ増加する。挿入部分に該当する要素は、null の格納番地である、0 にする。
- SVL は先頭に null 値を挿入する。

（これらの操作はシンプルかつシーケンシャルなので効率的に実行できる。）

This document is provided by JAXA.

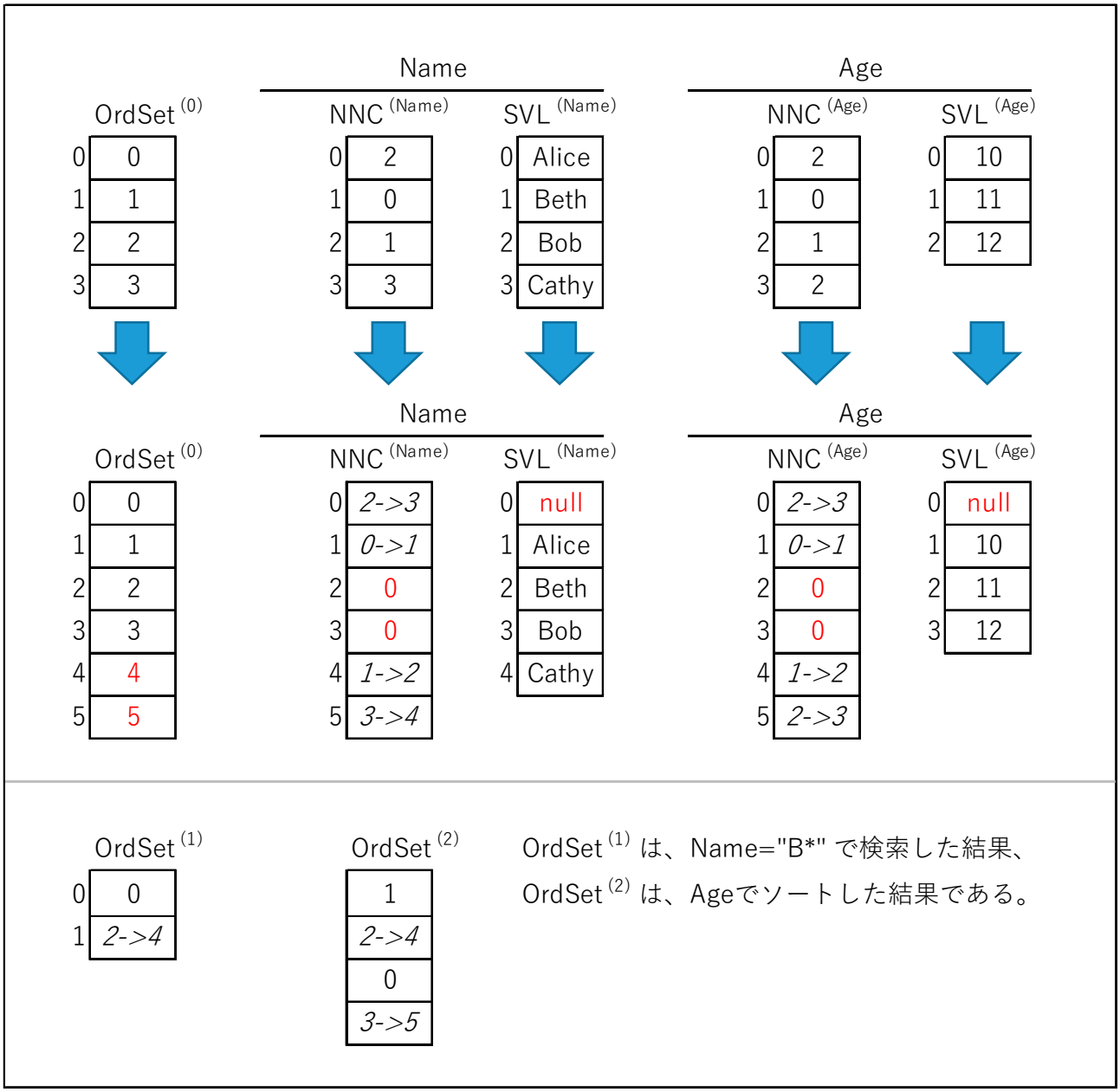


図 A-22.　レコード群の挿入による、各配列の変化

This document is provided by JAXA.

付録 8　実テーブル上のレコード群の削除のアルゴリズム

図 A-23 に示すテーブルから、図 A-24 に示すようにレコード群を削除するアルゴリズムを説明する。

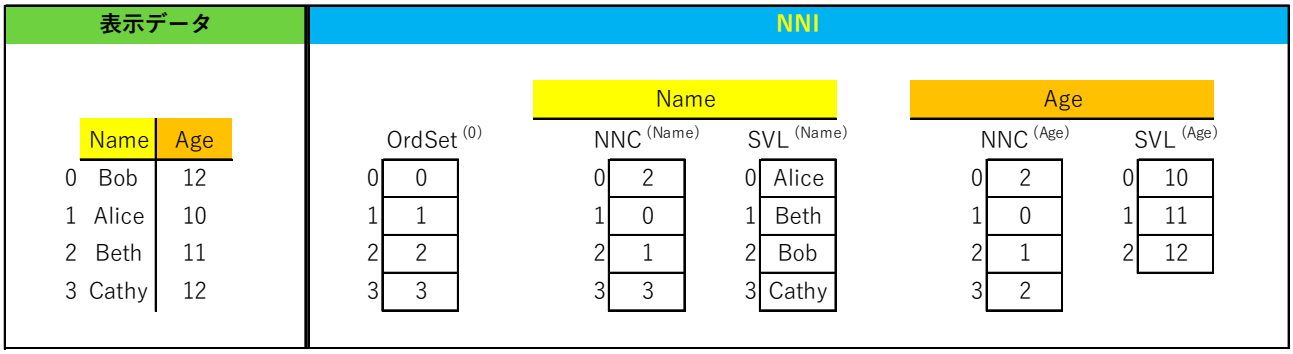


図 A-23.　レコード群の削除を行う前のテーブル

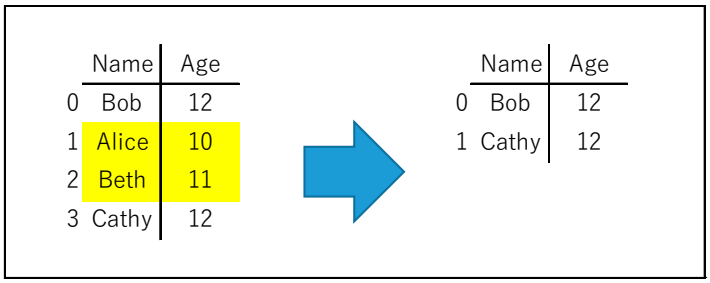


図 A-24.　例としてあげるレコード群の削除

図 A-25 に示す処理について以下に説明を加える。

- OrdSet (0) は要素数が 2 つ減るだけで、0 から始まるユニーク・連番であることに変わりは無い。
- OrdSet (0) 以外の OrdSet は、以下の扱い。
  - 削除位置(=1, 2)より小さい値を持つ要素はそのまま。
  - 削除位置(=1, 2)の値を持つ要素は削除。
  - 削除位置(=1, 2)より大きい値を持つ要素は、削除レコード数 (=2) だけデクリメント。
- NNC は、削除位置にある要素を削除。
- SVL は変化させない。
  （SVL は通常、データを Save する際にコンデンスするだけで十分。）

This document is provided by JAXA.

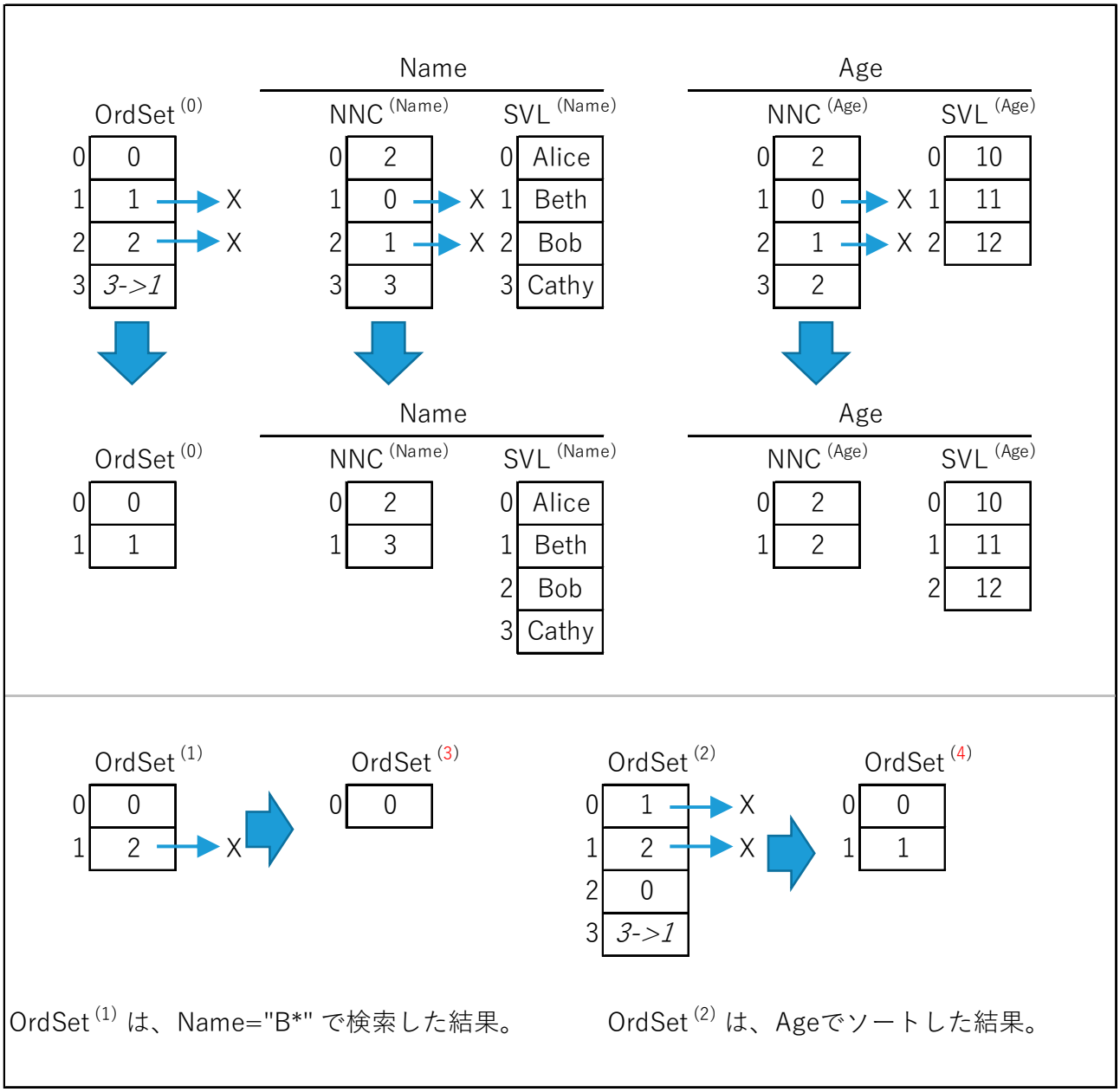


図 A-25. 例としてあげるレコード群の削除

This document is provided by JAXA.

付録 9　仮想ジョインテーブルの構造、読み出し方法、作成方法

A9.1　MAcm と SAcm

仮想ジョインテーブルは自然数インデックスで記述された 2 つの実テーブルからの写像として実現されるテーブルである。写像なのでデータ更新はできない。

図 A-30 は Name をジョインキーとして生成される仮想ジョインテーブル上の Master 側と Slave 側になる 2 つの実テーブルを示している。なお、Master 側テーブルとはレコードの順番が仮想ジョイン上に反映される側のテーブルを言う。Master 側でないテーブルを Slave 側テーブルと言う。Slave 側のテーブルはジョインキー値でソートされた上で仮想ジョインテーブルに組み込まれる。ここでは Table-A を Master 側として説明する。

Table-A

	Name	Point
0	Bob	10
1	Dolly	12
2	Cathy	11
3	Alice	12
4	Bob	10
5	Cathy	12

	OrdSet (A-0)
0	0
1	1
2	2
3	3
4	4
5	5

Name

	NNC (A-Name)
0	1
1	3
2	2
3	0
4	1
5	2

	SVL (A-Name)
0	Alice
1	Bob
2	Cathy
3	Dolly

Point

	NNC (A-Point)
0	0
1	2
2	1
3	2
4	0
5	2

	SVL (A-Point)
0	10
1	11
2	12

Table-B

	Name	Area
0	Amy	West
1	Bob	North
2	Dolly	South
3	Bob	East
4	Dolly	North
5	Alice	West

	OrdSet (B-0)
0	0
1	1
2	2
3	3
4	4
5	5

Name

	NNC (B-Name)
0	1
1	2
2	3
3	2
4	3
5	0

	SVL (B-Name)
0	Alice
1	Amy
2	Bob
3	Dolly

Area

	NNC (B-Area)
0	3
1	1
2	2
3	0
4	1
5	3

	SVL (B-Area)
0	East
1	North
2	South
3	West

図 A-30. 仮想ジョインテーブルの元になるテーブル A, B

図 A-31 は、図 A-30 の 2 つのテーブルから作成された仮想ジョインテーブルを示している。累計数配列 MAcm (Master-side Accumulation array)／SAcm (Slave-side Accumulation array) が Master 側／Slave 側に追加されている。なお、SAcm を使用可能にするため、ジョインキーとなるカラムの SVL がまず共通化される必要がある。SVL の共通化処理は既に付録 4 で説明した。

This document is provided by JAXA.

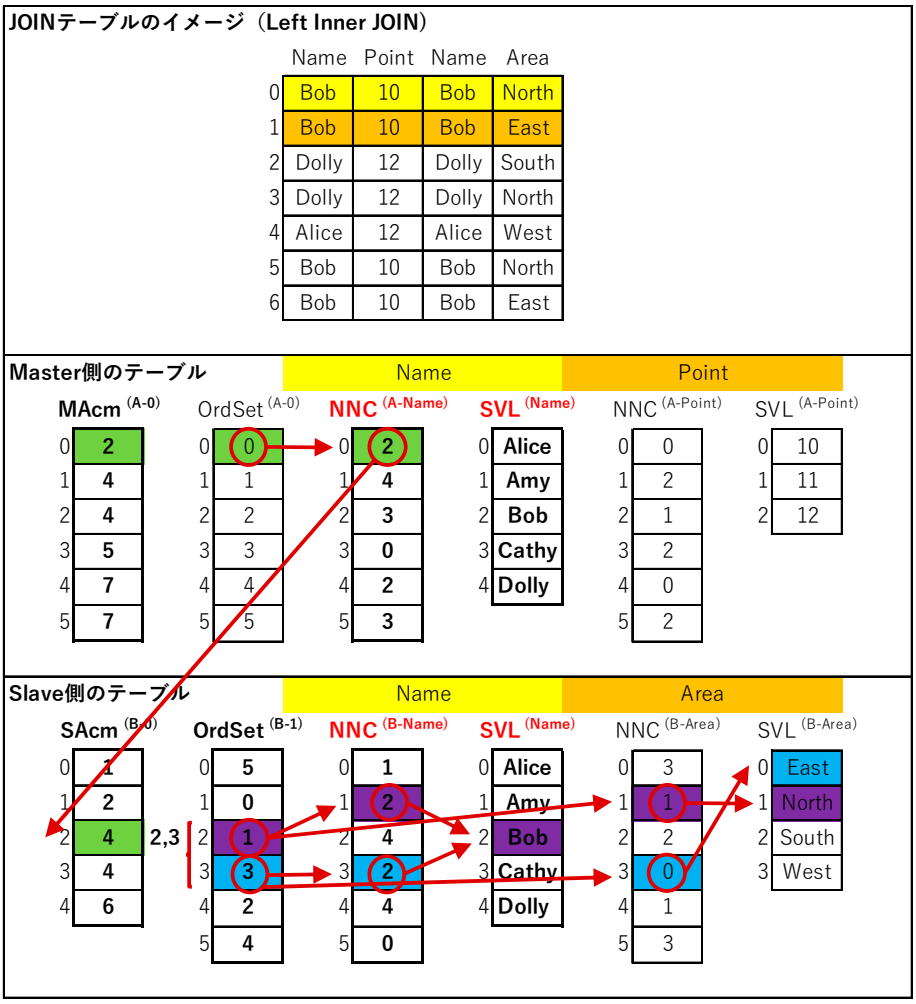


図 A-31．仮想ジョインテーブル

A9.2　MAcm (Master-side Accumulation array) について

図 A-31 の MAcm について説明する。なお、MAcm$^{(i)}$ は OrdSet$^{(Master-i)}$ と同じサイズになる。

まず、図 A-31 上段にあるジョインテーブルを見てみよう。ジョインテーブルは Master 側のテーブルの各レコードを以下の回数繰り返している。

レコード 0 (Bob)：　2 回
レコード 1 (Dolly)：　2 回
レコード 2 (Cathy)：　0 回
レコード 3 (Alice)：　1 回
レコード 4 (Bob)：　2 回
レコード 5 (Cathy)：　0 回

これを累計数化すると、{2, 4, 4, 5, 7, 7} が得られる。これが MAcm となる。
MAcm について以下の式が成り立つ。

Master テーブルのレコード i がジョインテーブル内で出現する位置 ＝MAcm[i-1]　… 式(A-1)
Master テーブルのレコード i のリピート回数 ＝MAcm[i] – MAcm[i–1]　… 式(A-2)

This document is provided by JAXA.

MAcm は二つの役割を果たす。

- 式(A-1)を使って $I_{(join)}$ から $I_{(master)}$ を特定する。
- 式(A-2)を使って $I_{(master)}$ のリピート回数を求める。

A9.3 SAcm (Slave-side Accumulation array) について

図 A-31 の SAcm について説明する。$SAcm^{(i)}$ は $SVL^{(JoinKey)}$ と同じサイズになる。Slave 側のテーブルのレコード数は、ジョインキーとなる $SVL^{(Name)}$ の各値について以下の回数となる。(図 A-30 の Table-B を参照して確かめられたい)

$SVL^{(Name)}$ [0] (Alice) : 1 回
$SVL^{(Name)}$ [1] (Amy) : 1 回
$SVL^{(Name)}$ [2] (Bob) : 2 回
$SVL^{(Name)}$ [3] (Cathy) : 0 回
$SVL^{(Name)}$ [4] (Dolly) : 2 回

これを累計数化すると、{1, 2, 4, 4, 6} が得られる。これが SAcm となる。
SAcm について以下の式が成り立つ。

SVL[i] の値が Slave 側のテーブル内で出現する位置 = SAcm[i-1] ... 式(A-3)
SVL[i] の値が Slave 側のテーブル内で出現する回数 = SAcm[i] – SAcm[i–1] ... 式(A-4)

SAcm は以下の役割を果たす。

$I_{(master)}$ から $NNC^{(A\text{-}Name)}$ を経由して式(A-3)、式(A-4)を使い、同一の SVL 値を持つ $I_{(slave)}$ 群を特定する。

A9.4 仮想ジョインテーブルのレコードの読み出し方法

仮想ジョインテーブルのレコードを読み出すには、仮想ジョインテーブルのレコード番号 $I_{(join)}$ を与えて、Master 側のテーブルのレコード番号 $I_{(master)}$ と Slave 側のテーブルのレコード番号 $I_{(slave)}$ を得られば良い。それを、$I_{(join)} = 1$ のケースを例にして説明する。(図 A-31 参照)

1. Step-1. $I_{(master)}$ の特定
   MAcm[j] > $I_{(join)}$ を満たす最小の j を求める。(バイセクション法が使える)
   MAcm[0] (= 2) > $I_{(join)}$ (= 1) なので、 j = 0 である。
   $I_{(master)}$ = j = 0 となる。

2. Step-2. offset を求める
   レコード Master[$I_{(master)}$] に対応して複数のレコード Slave[$I_{(slave)}$] が出現し得るが、offset とはその中の 0 から始まる順位である。
   式 (1)により、MAcm[$I_{(master)}$ - 1] = MAcm[0-1] (= 0) がレコード： Master[$I_{(master)}$] の、ジョインテーブル内での開始位置である。以下の式で計算できる。
   offset = $I_{(join)}$ - MAcm[$I_{(master)}$ - 1] = 1 - 0 = 1

This document is provided by JAXA.

3. Step-3. base を求める

レコード Master[$I_{(master)}$] に対応して複数のレコード Slave[$I_{(slave)}$] が出現し得るが、base とはその開始位置である。以下の式で計算できる。

base = SAcm[$NNC^{(A\text{-}Name)}$ [$OrdSet^{(A\text{-}0)}$ [$I_{(master)}$]] -1]

= SAcm[$NNC^{(A\text{-}Name)}$ [$OrdSet^{(A\text{-}0)}$ [0]] - 1]

= SAcm[$NNC^{(A\text{-}Name)}$ [0] - 1]

= SAcm[2 - 1]

= SAcm[1]

= 2

4. Step-4. $I_{(slave)}$ を求める

$I_{(slave)}$ = base + offset = 2 + 1 = 3

5. Step-5. JOIN [$I_{(join)}$] を確定する。

JOIN [$I_{(join)}$ = 1]　= Master[$I_{(master)}$] & Slave[$I_{(slave)}$]

= Master[0] & Slave[3]

= (Bob, 10) & (Bob, East)

A9.5　仮想ジョインテーブルの作成ステップ

1. SVL の共通化処理と NNC の更新

ジョインキーとなるカラムの SVL を共通化（4.1 参照）する。

2. $OrdSet^{(B\text{-}1)}$ 、SAcm を作成する。
3. MAcm を作成する。

A9.5.1　SVL の共通化処理と NNC の更新

SVL の共通化処理（4.1 参照）を行う。（図 A-32）

This document is provided by JAXA.

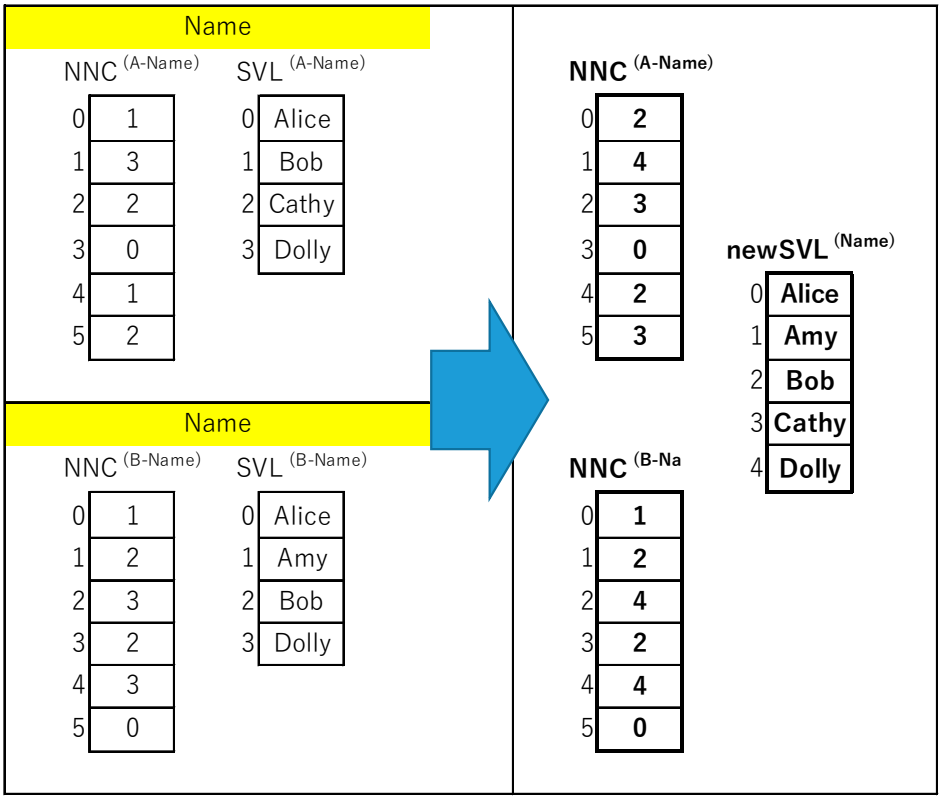


図 A-32. SVL の共通化処理の前後

図 A-32 に示すように、Conv 配列を参照しながら、NNC を更新する。

A9.5.2 OrdSet (B-1)、SAcm を作成する

図 A-33 に示すように、Slave 側のテーブルをソートし、OrdSet$^{(B-1)}$ を作成する。ソートを行う際に使用する Aggr 配列（ソートのアルゴリズムを参照）が、ソート後、そのまま SAcm になる。

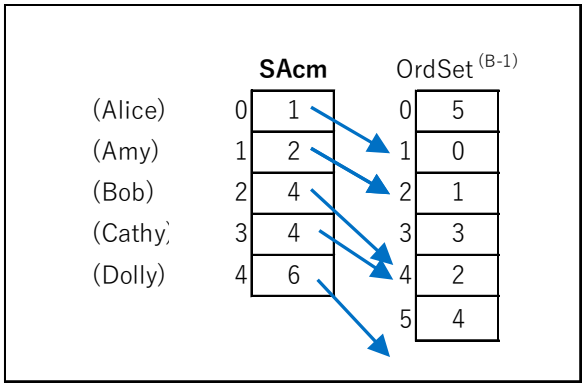


図 A-33. Slave 側をジョインキーでソート

A9.5.3 MAcm を作成する

図 A-34 中の Count(Master[i]) は、Master 側のテーブルの各レコードの出現回数である。この出現回数は以下の式で求められる。

This document is provided by JAXA.

$$\mathrm{Count}(\ \mathrm{Master}[i]\ ) = \mathrm{SAcm}[\ \mathrm{NNC}^{(\mathrm{A\text{-}Name})}[\ \mathrm{OrdSet}^{(\mathrm{A\text{-}0})}[i]\ ] - \mathrm{SAcm}[\ \mathrm{NNC}^{(\mathrm{A\text{-}Name})}[\ \mathrm{OrdSet}^{(\mathrm{A\text{-}0})}[i]\ ] - 1\ ]$$

この式を使って、Count(Master[i]) 配列を作成し、それを累計数化すると MAcm ができる。

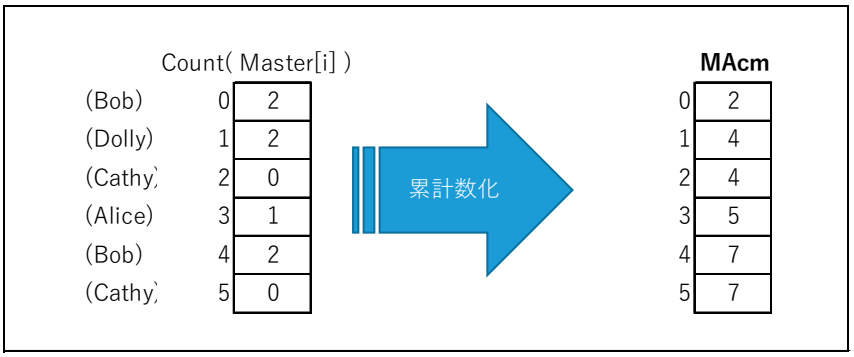


図 A-34.　MAcm を作る

以上で、ジョインテーブルを構成するすべての成分を構成、更新することができた。

This document is provided by JAXA.

付録 10　仮想ジョインテーブル上の検索のアルゴリズム

仮想ジョインテーブルの検索は、Master 側もしくは Slave 側のテーブルを検索して、再度ジョインを行うことで実現する。検索のために再度のジョインを行う際には、SVL の共通化とその関連の処理はすでに済んでいるので行わなくて良い。

この方式で正しく検索できることが保証されるのは以下である。

第 1　Master 側テーブルのカラム群のみに対する検索
第 2　Slave 側テーブルのカラム群のみに対する検索
第 3　上記第 1、第 2 の検索をさらに AND で結合する検索

第 1 の検索の例を示そう。図 A-31 の仮想ジョインテーブルを、Master 側のカラムである Point に対して、“Point >= 11” で検索する。検索により、$\mathrm{OrdSet}^{(A\text{-}0)} = \{0, 1, 2, 3, 4, 5\}$ から $\mathrm{OrdSet}^{(A\text{-}1)} = \{1, 2, 3, 5\}$ を得る。$\mathrm{OrdSet}^{(A\text{-}1)}$ を使って改めてジョインを行うと図 A-35 のようになる。

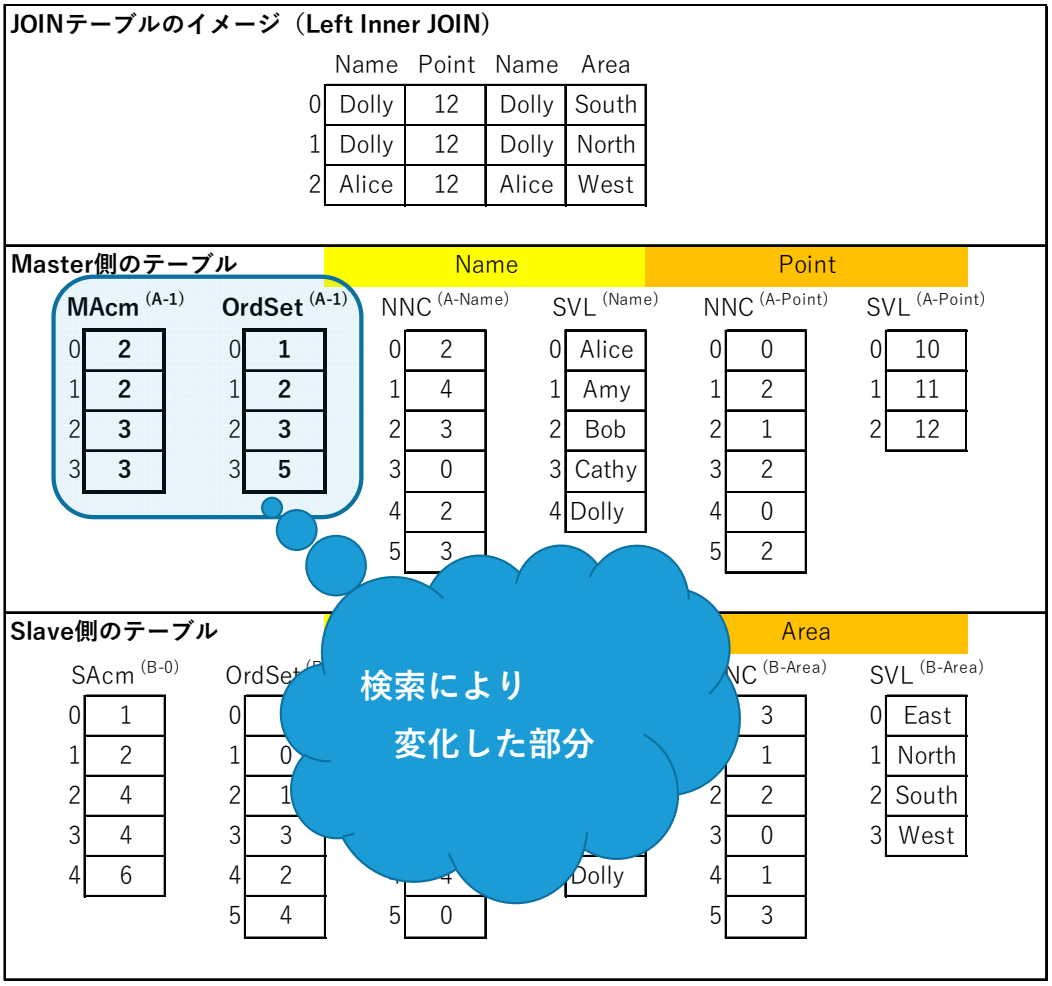


図 A-35. “Point >= 11” で検索した結果

This document is provided by JAXA.

付録 11　仮想ジョインテーブル上のソートのアルゴリズム

仮想ジョインテーブルのソートも検索と同様に、Master 側もしくは Slave 側のテーブルをソートして、再度ジョインを行うことで実現する。ソートの安定性を利用して、複数カラムのソートは各カラムのソートをカスケードすれば良い。ソートのために再度のジョインを行う際には、SVL の共通化とその関連の処理はすでに済んでいるので行わなくて良い。

仮想ジョインテーブルは Master 側のテーブルのレコード順しか反映できないので、Slave 側のカラム群でソートしたい場合は、Slave 側を Master 側と見なしてソートすることになる。Master 側と Slave 側の両方のカラムでソートを成立させることは特別な場合を除いてできない。

例を示そう。図 A-31 の仮想ジョインテーブルを“Point” でソートする。$\mathrm{OrdSet}^{(A\text{-}0)} = \{0, 1, 2, 3, 4, 5\}$ をソートして $\mathrm{OrdSet}^{(A\text{-}1)} = \{0, 4, 2, 1, 3, 5\}$ を得た後、再度のジョインを行うと図 A-36 のようになる。

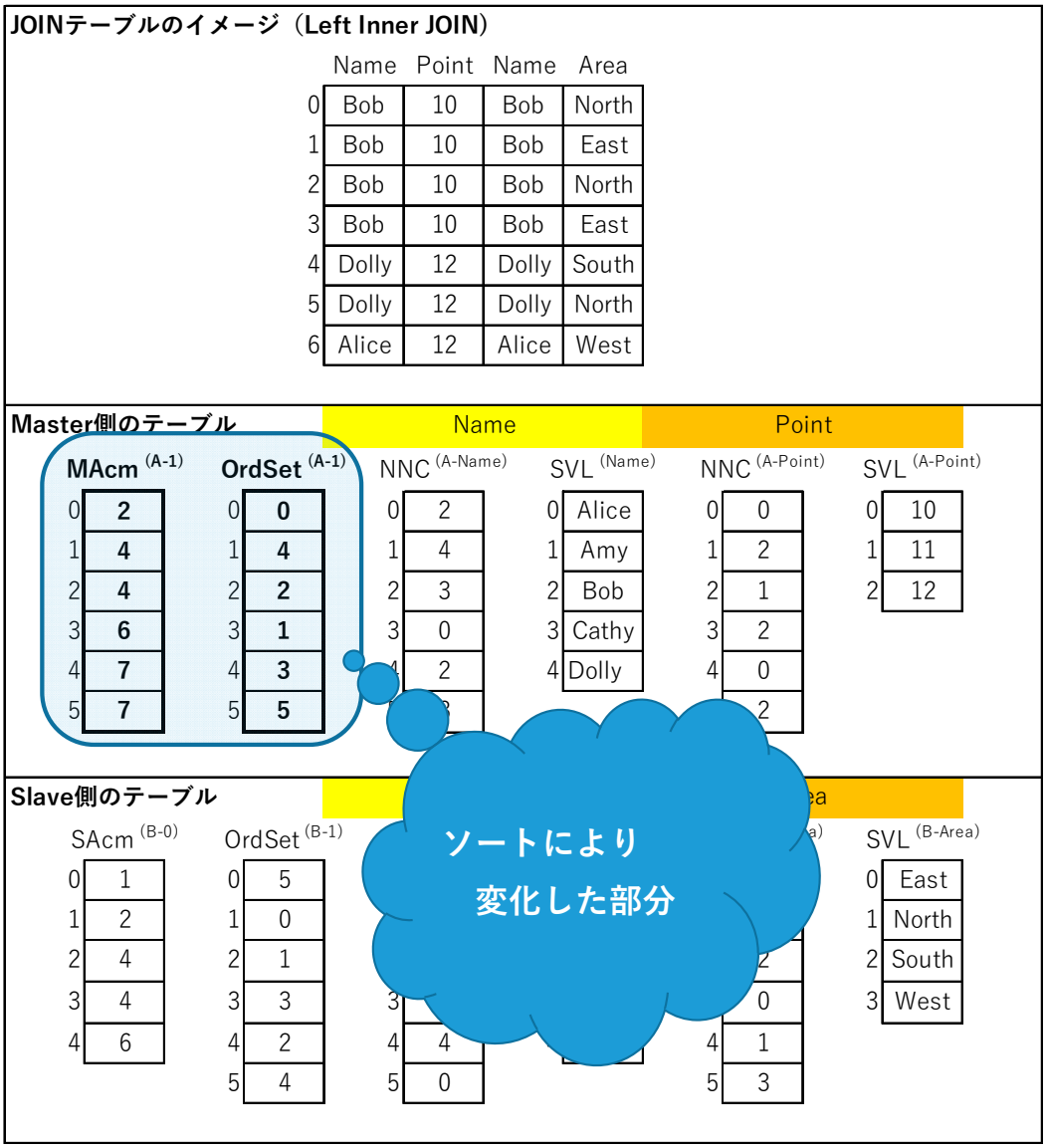


図 A-36.　“Point” でソートした結果

This document is provided by JAXA.

付録 12　仮想ジョインテーブル上の集計のアルゴリズム

図 A-37 に示すテーブル A とテーブル B の仮想ジョインテーブルを集計してみよう。ここでは、Drink 毎、Type 毎に、Price の合計を求める。次元は Drink と Type、測度は Price である。

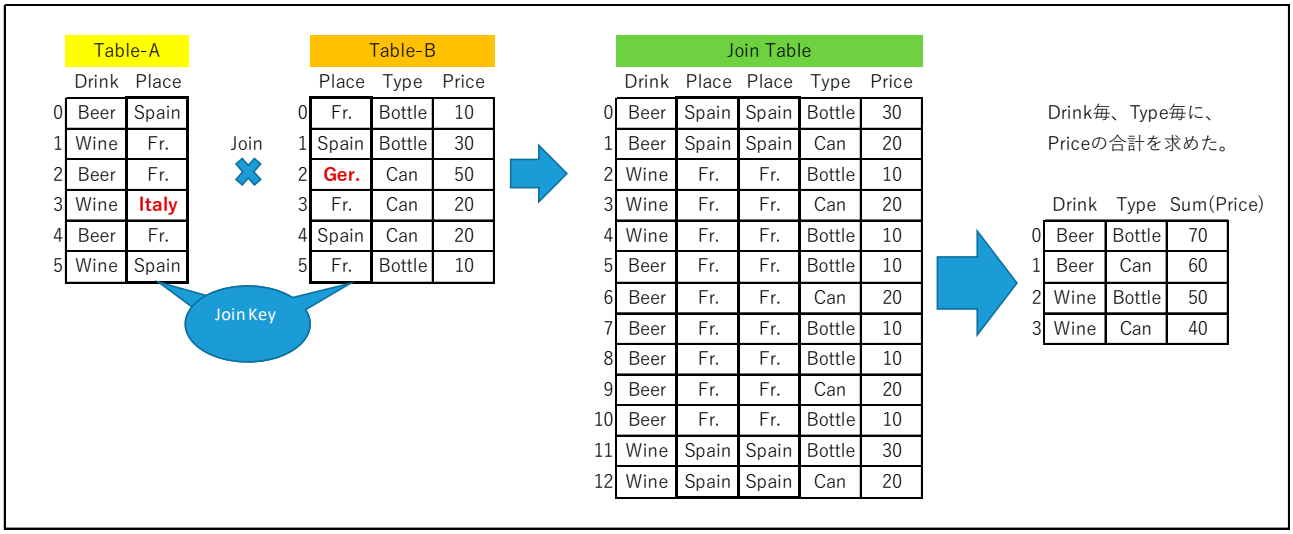


図 A-37. 行おうとする集計

Step-1. まず集計の次元、次にジョインキーでソートする。

この操作で、ジョインキー毎、集計の次元毎に並び替えられる。（図 A-38）

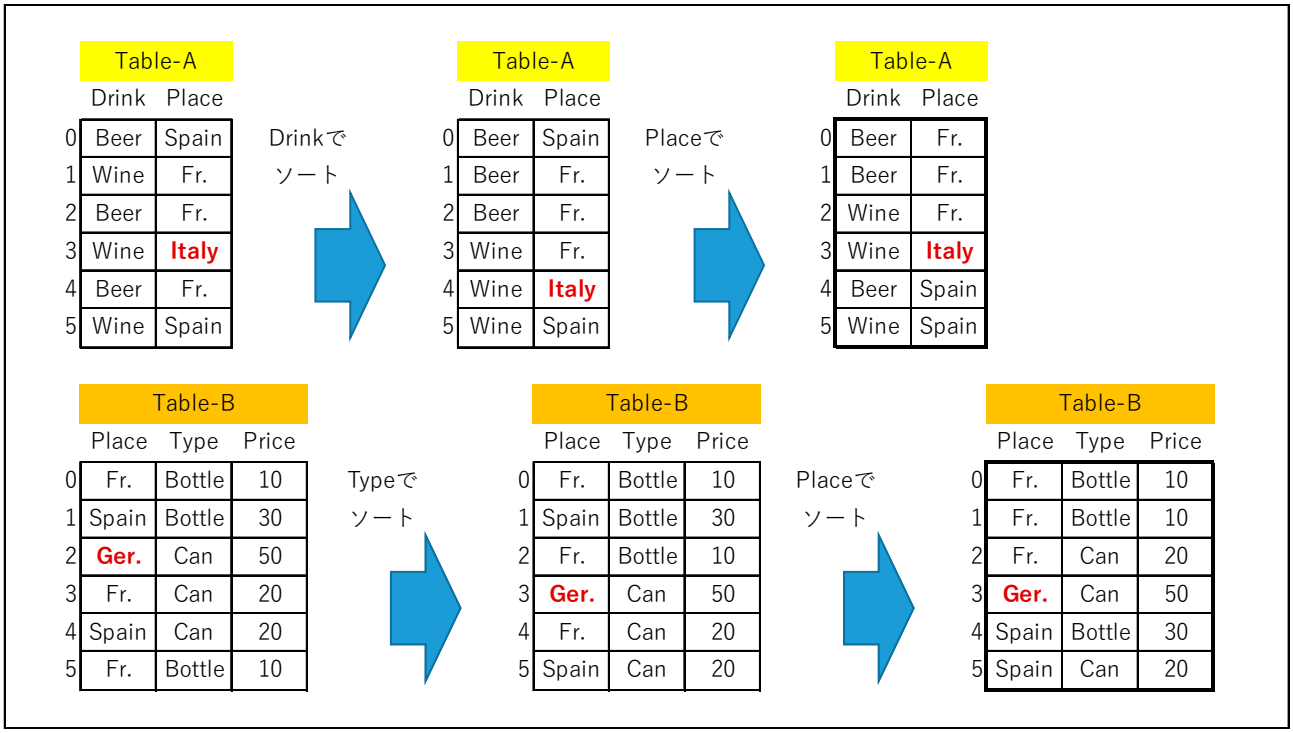


図 A-38. まず集計の次元、次にジョインキーでソート

This document is provided by JAXA.

Step-2. ジョインキー値が同一の範囲毎に、Master/Slave それぞれの中で集計を行い、その積を作る。

図 A-39A から図 A-39D に示すように、ジョインキー値毎に集計を作る。ジョインキー値毎の集計結果は順次足し込んで行く。（図 A-40）

次元である Drink、Type の SVL $^{(Drink)}$ 、SVL $^{(Type)}$ は付録 3 で述べたようにそのまま集計結果テーブルに引き継がれるので、次元の作成効率は高い。

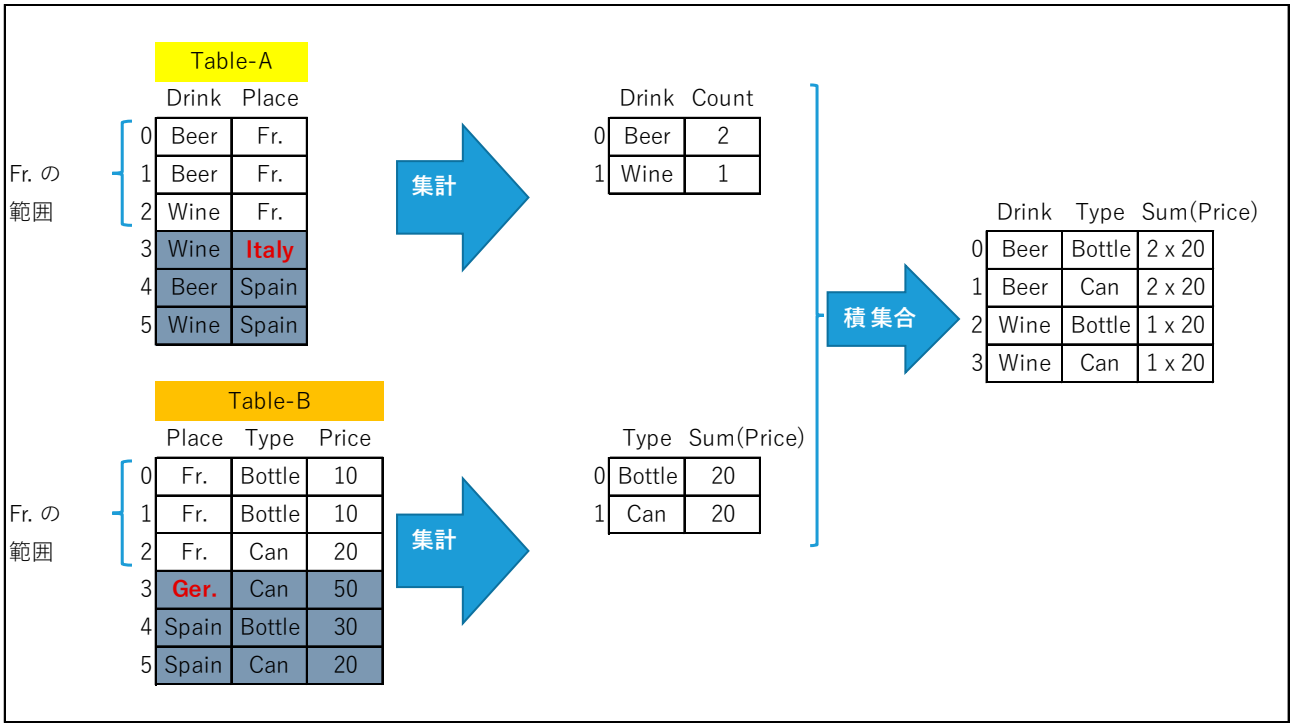


図 A-39A. ジョインキー="Fr." の集計

Table-A にGer.がなく、処理不要。

図 A-39B. ジョインキー="Ger." の集計

Table-B にItalyがなく、処理不要。

図 A-39C. ジョインキー="Italy" の集計

This document is provided by JAXA.

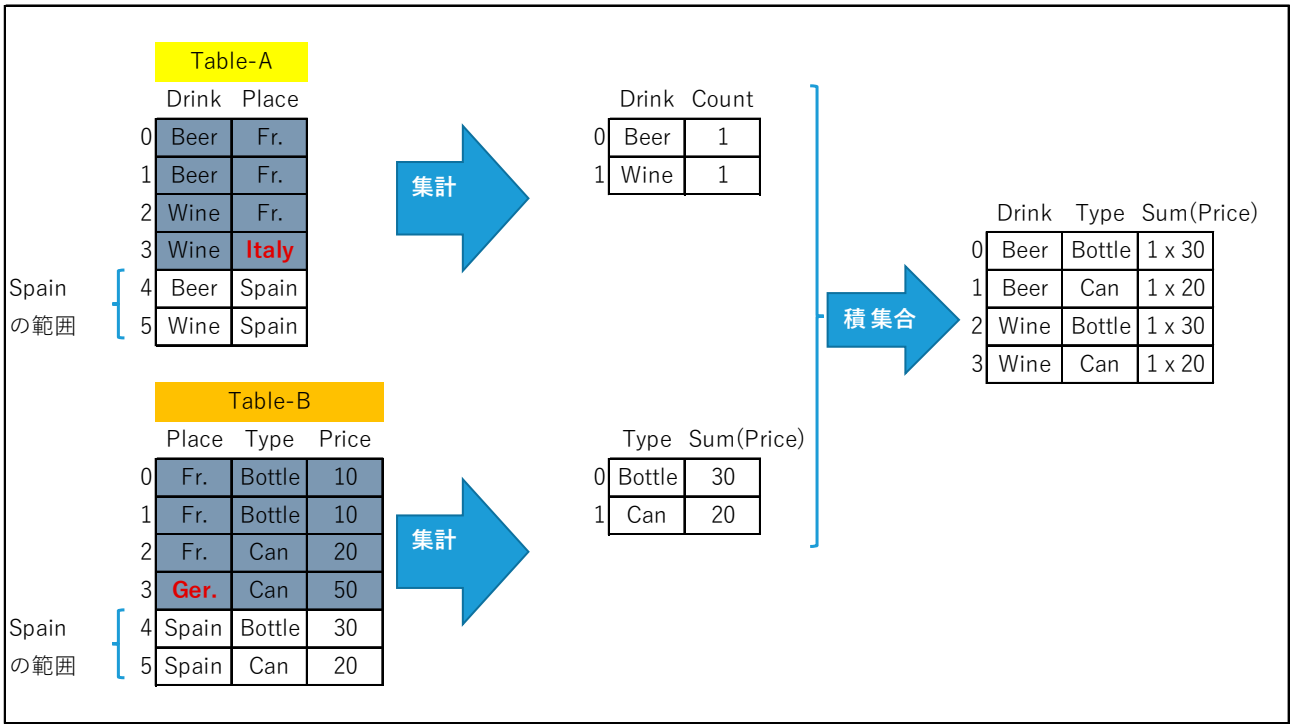


図 A-39D. ジョインキー="Spain" の集計

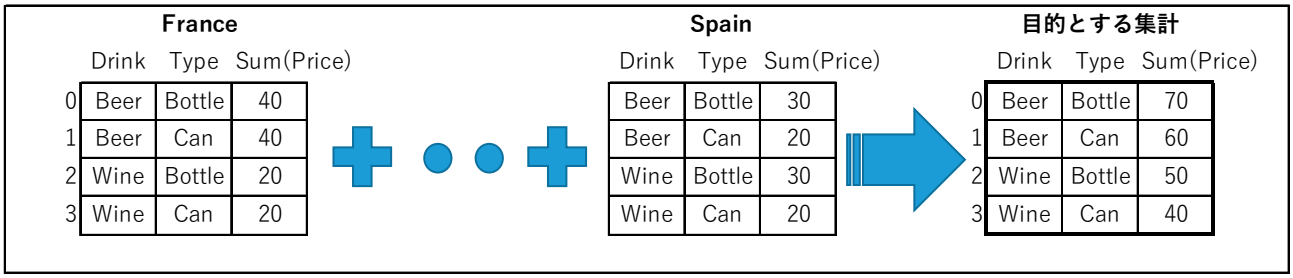


図 A-40. 各ジョインキー値の集計を足し込む

This document is provided by JAXA.

付録 13　カラム転送のアルゴリズム

仮想ジョインテーブルのカラム転送とは、Slave 側のテーブルのカラムを、ジョインキーを経由して Master 側のテーブルにコピーする機能である。転送されるカラムの SVL はそのままコピーすれば良く、NNC だけを再計算すれば完了するので高速である。

サブテーブル間の仮想ジョインは可能であるため、カラム転送も Master 側および Slave 側のサブテーブル間で成立する。ここでは“Point >= 11”という検索でできた仮想ジョインのサブテーブルを使ってアルゴリズムを説明する。

図 A-41 は、仮想ジョインテーブルを構成する元になった Master 側（Table-A）と、Slave 側（Table-B）を示している。これをジョインすると、図 A-42 に示す仮想ジョインテーブルができる。

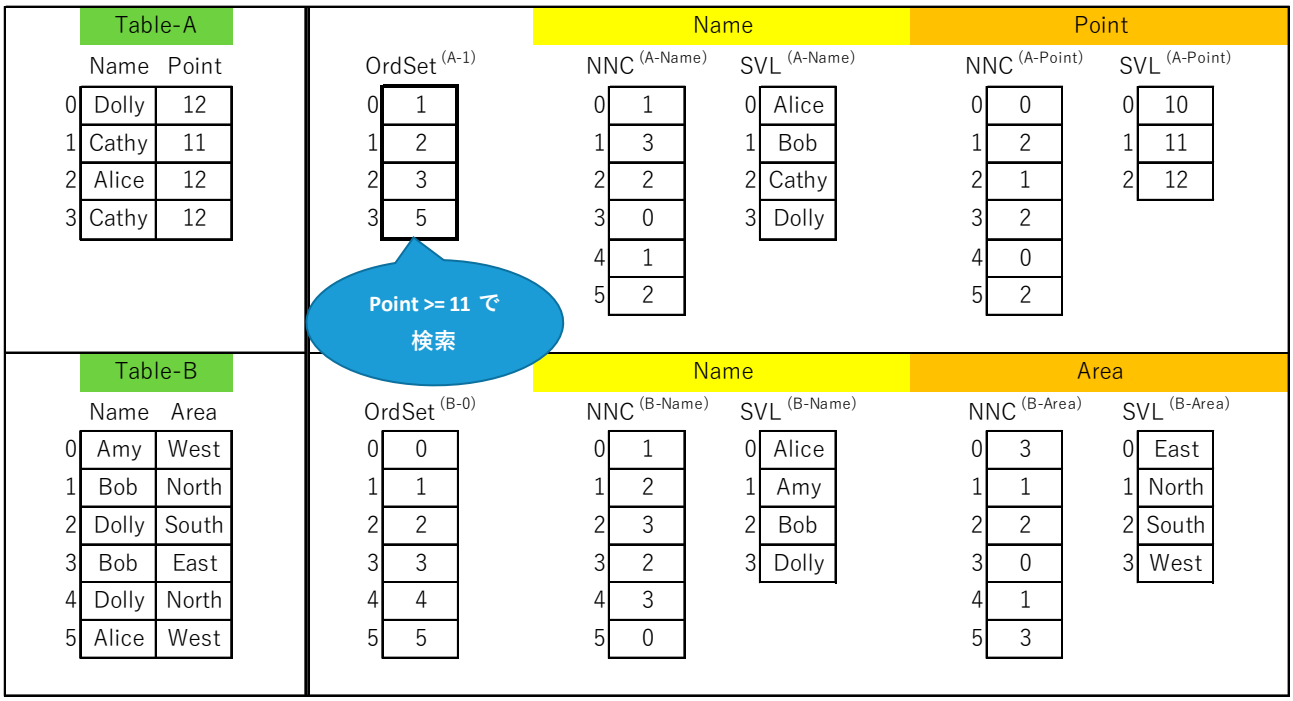


図 A-41．“Point >= 11”の検索でできたサブテーブル（A）と、テーブル（B）

This document is provided by JAXA.

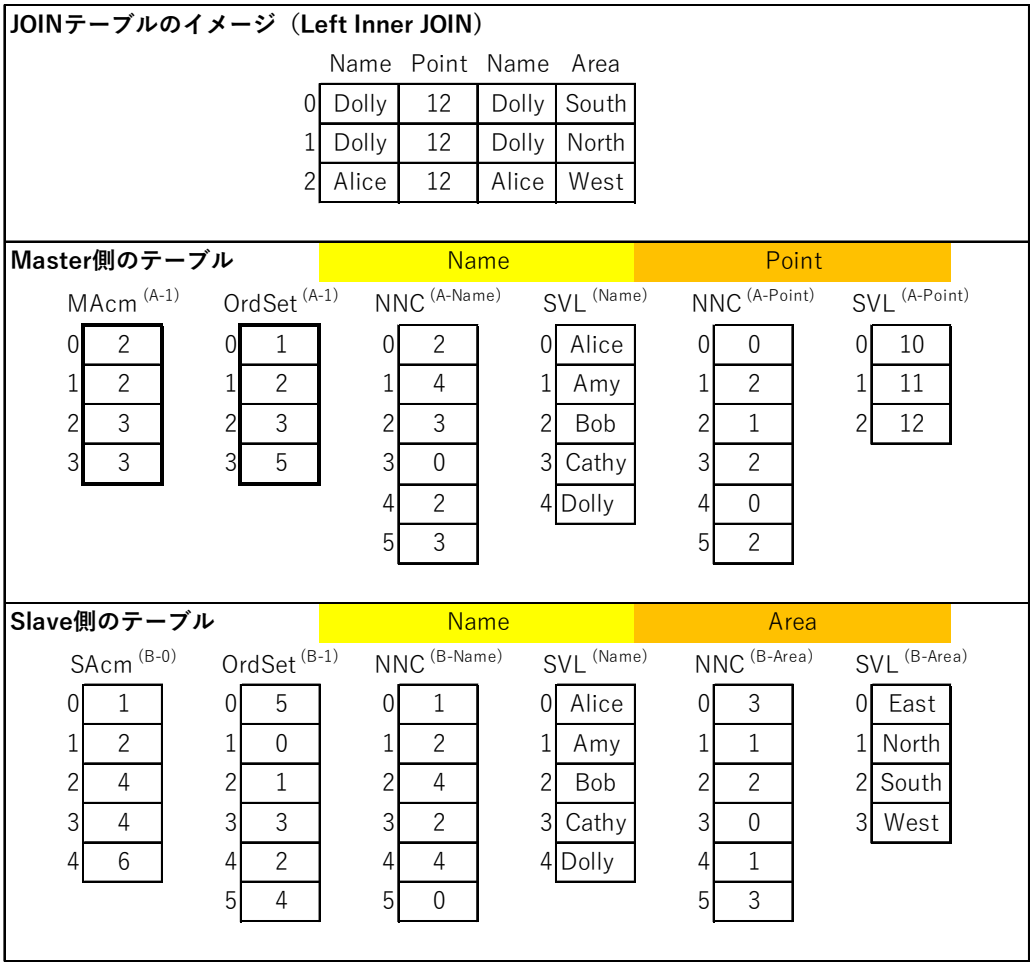


図 A-42. 例とする仮想ジョインテーブルの内部

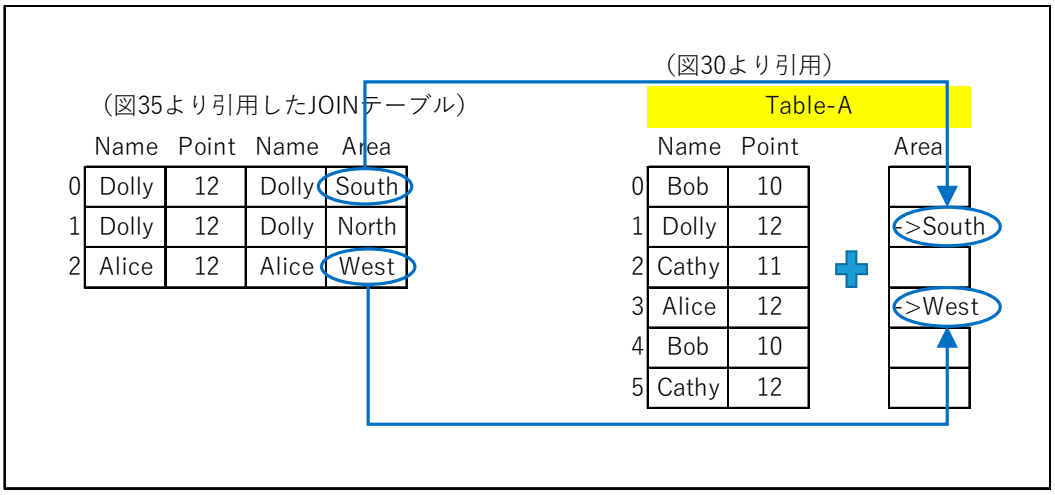


図 A-43. 例とするカラム転送

図 A-43 は、カラム転送の見かけのイメージを説明している。カラム転送により、Master 側のテーブルに、新たに“Area”カラムが追加される。そのカラムの要素は、Slave 側のテーブルからジョインキーを経由して転送されてきたものだ。ジョインキーがマッチしないケースでは転送されない。1 つのジョイ

This document is provided by JAXA.

ンキーで複数の Slave 側のテーブルのレコードが対応するとき、表示順で先頭の Slave 側のレコードの保持する値が転送される。

項目転送の操作は下記のコードで記述できる。図 A-44 はそれをチャートで表している。なお、図 A-42 の $MAcm^{(A-1)} = \{2, 2, 3, 3\}$ と $OrdSet^{(A-1)} = \{1, 2, 3, 5\}$ の組合せは $OrdSet^{(A-2)} = \{1, 3\}$ と等価であるため、下記のコード及び図 A-44 では $OrdSet^{(A-2)}$ を用いて説明している。（当初の SVL 中には null 値が登録されていない場合の説明であることに注意）

$$NNC^{(A\text{-}Area)}[\,OrdSet^{(A\text{-}2)}[i]\,] := 1 + NNC^{(B\text{-}Area)}[\,OrdSet^{(B\text{-}1)}[\,SAcm[\,-1 + NNC^{(A\text{-}Name)}[\,OrdSet^{(A\text{-}2)}[i]\,]\,]\,]$$

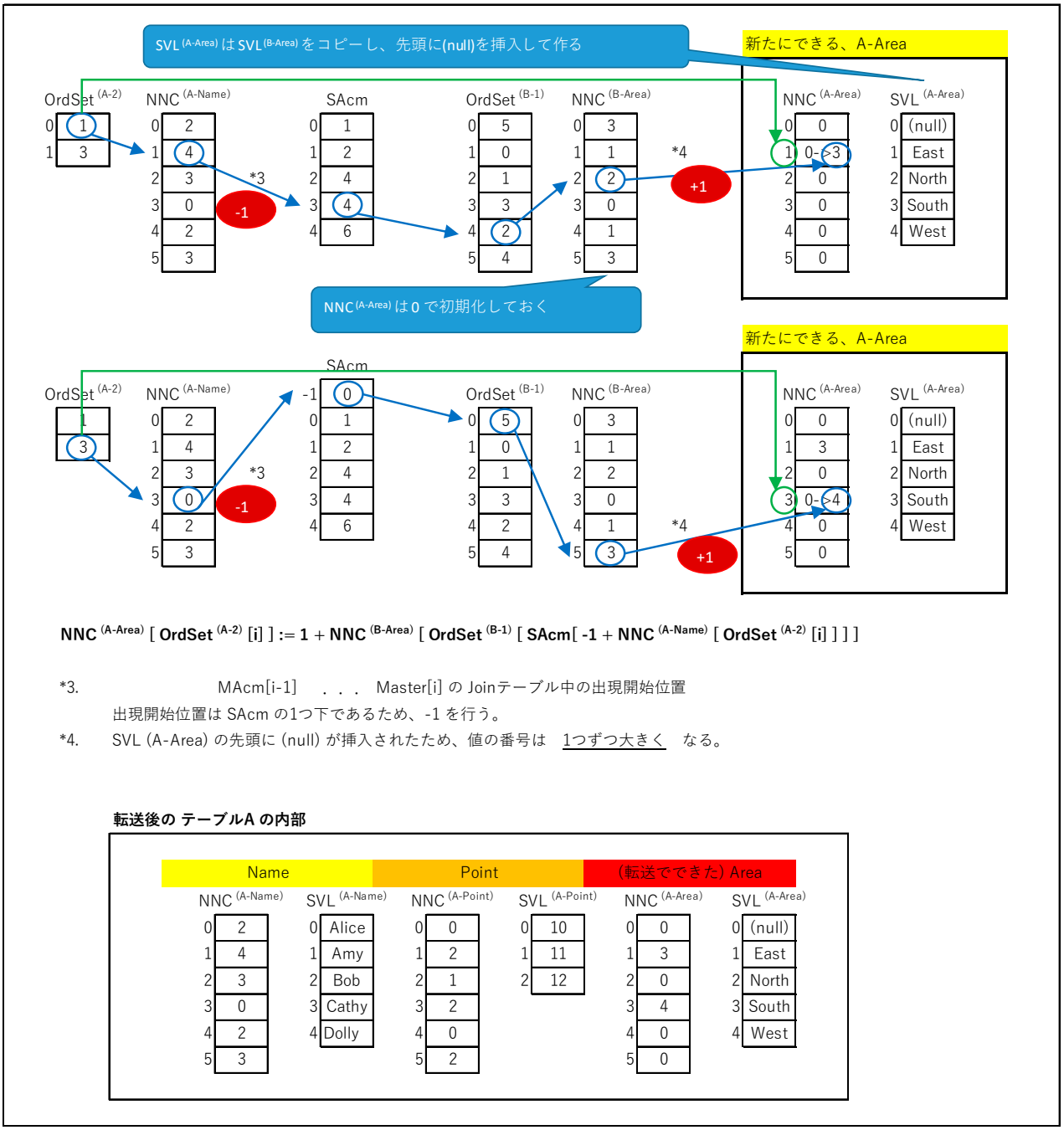


図 A-44.　カラム転送の実際

This document is provided by JAXA.

付録 14 ジョインにマッチした集合、しなかった集合の取出しアルゴリズム

仮想ジョインテーブルはジョインにマッチした集合、外れた集合を容易に取り出すことができる。取り出された集合は、Master 側のテーブルもしくは Slave 側のテーブルに新たな OrdSet として登録される。アルゴリズムを見ると高速に実行できることが分かる。

この機能はいろいろな用途があるが、例えば高速な相関副問合せを実現するのに使われる。

■ Master 側のマッチする集合 OrdSet (A-2) 、しない集合 OrdSet (A-3) の摘出

Master 側のテーブルにジョインにマッチした集合、しなかった集合を転送するアルゴリズムを図 A-45 に示す。OrdSet から NNC を経由して SAcm をアクセスし、0 件以上のレコードが Slave 側に存在しているかどうかを調べることでジョインキーにマッチした集合（しなかった集合）を作成できる。

単なるスキャンで作成できるため高速である。

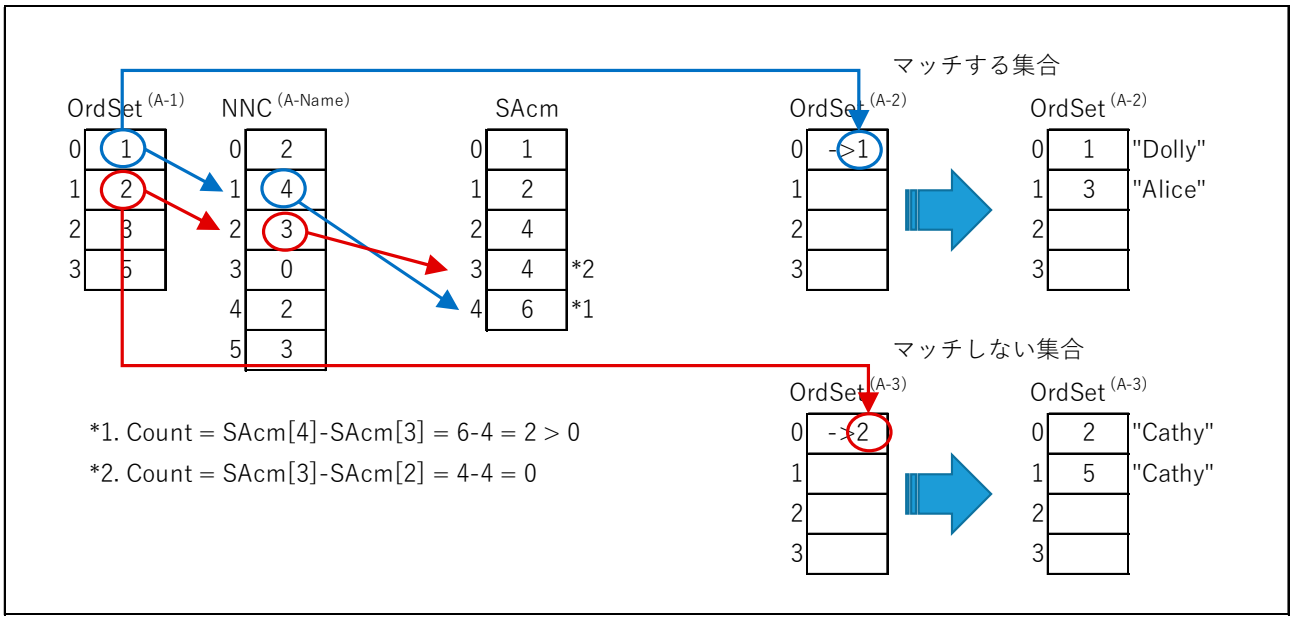


図 A-45. Master 側のテーブル用のジョインにマッチした（しなかった）集合を作るアルゴリズム

■ Slave 側のマッチする集合 OrdSet (B-2) 、しない集合 OrdSet (B-3) の摘出

Slave 側のテーブルにジョインにマッチした集合、しなかった集合を転送するアルゴリズムを図 A-46 に示す。Slave 側のテーブルから Master 側のテーブルのレコードを参照するパスは仮想ジョインテーブル内に用意されてはいない。従って準備作業を必要とする。

まず、Master 側の OrdSet から、Master 側の NNC を経由して Flags 配列をマークする。（マークしてそのジョインキー値に対応する Master 側のレコードがあることを記録する。）Flags 配列のサイズはジョインキーの共通化された SVL のサイズと同じである。

次に Slave 側の OrdSet から Slave 側の NNC を経由して Flags 配列を参照し、Master 側に対応するレコードがあるかどうかを検出する。

この処理は Master 側の処理より時間がかかるが、シンプルな処理であり、高速である。

This document is provided by JAXA.

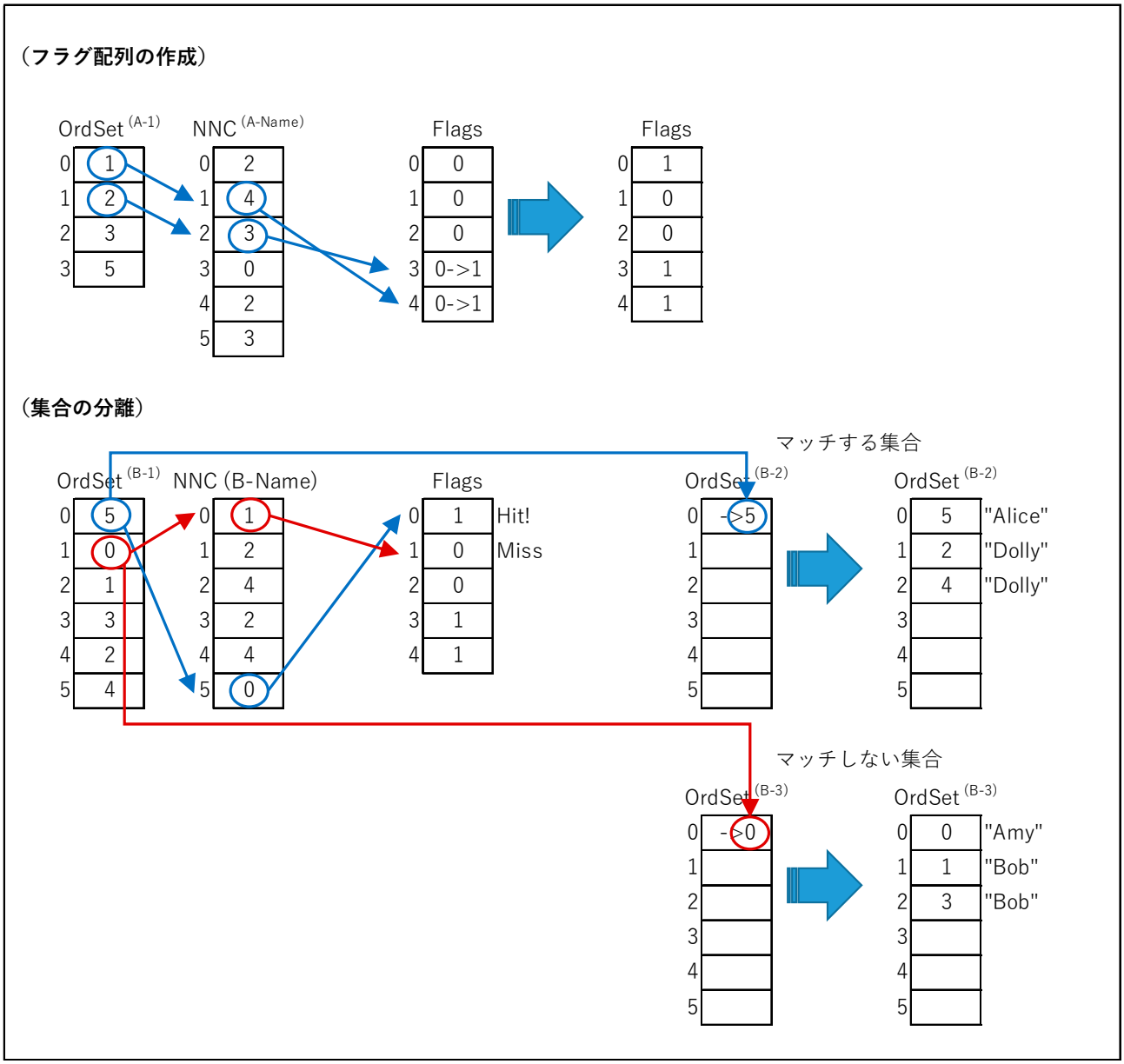


図 A-46．Slave 側のテーブル用のジョインにマッチした（しなかった）集合を作るアルゴリズム

This document is provided by JAXA.