DBF - Optimizer

ASL, Sargable Predicates

Search Argumentable Predicates

- Beispiel: C = const4

```
NEXT(predicate); NEXT(predicate);
NEXT(predicate); …
```

(Data Manager kann predicate verarbeiten)

Non Search Argumentable Predicates

- Beispiel: C1 + C2 = 17

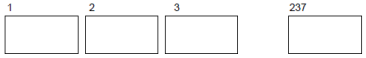
```
NEXT; RDS-check(predicate); NEXT; RDS-
check(predicate); …
```

(DM muss check dem RDS überlassen, liefert also zuviel)

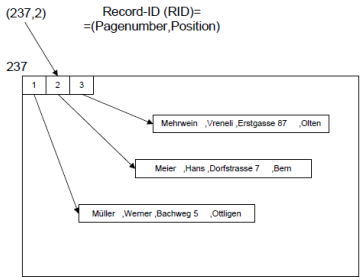
"Physical" Layout

- File Management System spielt dem DB System z.B. linearen Adressraum vor, in einem File

Pagennummern



Record Adressierung:



Indexes

- Erzwingen der Eindeutigkeit von Werten in Tabelle (**UNIQUE INDEX**)
- Zugriffshilfe auf Tabellenzeilen (**kann** auch **unique** sein, muss aber nicht)
- Beispiel: Index auf Attribut Name:

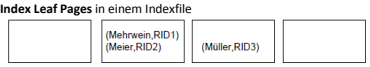
  Index Einträge:

```
(Mehrwein,(237,3)), (Meier,(237, 2)),
(Müller,(237,1)) ,…
```

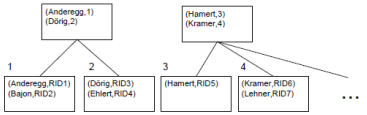
  Allgemeiner:

```
(Mehrwein, RIDp), (Meier, RIDq), (Müller,
RIDr), …
```

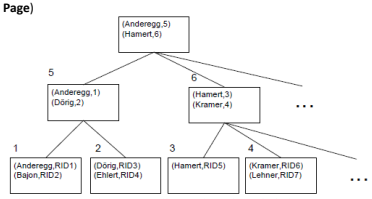
- Index-Einträge: **lexikografisch** geordnet abgelegt in Menge von **Index Leaf Pages** in einem Indexfile



- Pages müssen auf Disk nicht physisch hintereinander liegen, sind aber verpointert, so dass das System sich der lexikografischen Ordnung entlang hangeln kann (vorwärts und rückwärts) in einem **Leaf Page Scan**
- Erster Indexeintrag jeder Leaf Page, zusammen mit einem Pointer auf diese Page, wird herausgezogen und in einer zweiten Schicht von Pages versammelt



- … wird fortgesetzt, bis nur noch eine Page notwentig ist (**Root Page**)



- Alle Indexpages liegen im selben Index-File, in beliebiger Reihenfolge, nach einer Reorganisation idealerweise zum Beispiel in top-down-left-right-Ordnung
- Logische Ordnung der Pages untereinander → **B+ Baum**
  - "B" für balanced (kürzester Weg von Root Page zu Leaf Page ist immer gleich lang, nämlich Anzahl Levels - 1)
  - "+" bedeutet, dass Pages jeden Levels je unter sich verpointert sind, gemäss der lexikographischen Reihenfolge der (Attribut-) Werte, meist jeweils vorwärts und rückwärts

Index über mehrere Attribute

- Wird ein Index über zwei Attribute definiert

```
CREATE INDEX Xperson1 ON Person (Name,
Vorname)
```

dann haben die Index Leaf Page Einträge die Form

```
(Ambach,Heinz,RID1), (Meier,Hans,RID2),
(Meier,Urs,RID3), etc.
```

die lexikographische Ordnung ist dann definiert wie im Telefon-buch, nämlich

```
(x,y) < (a,b) ist definiert als
(x<a)∨(x=a ∧ y<b)
```

allgemeiner ist

```
(x1,x2,…,xn)<(a1,a2,…,an)
```

rekursiv definiert als

```
(x1<a1)∨(x1=a1 ∧ (x2,…,xn)<(a2,…,an))
```

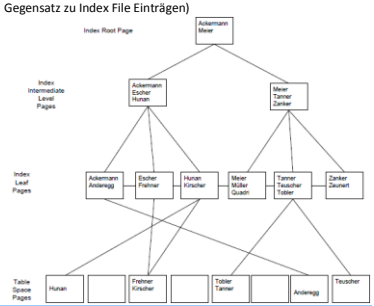
und natürlich mit der Verankerung (x1) < (a1) als x1 < a1

- Kommen in einem non-unique Index Werte von Indexattributen mehrfach vor, dann wird in den Index Leafpages ein Wert oft nur einmal gespeichert:

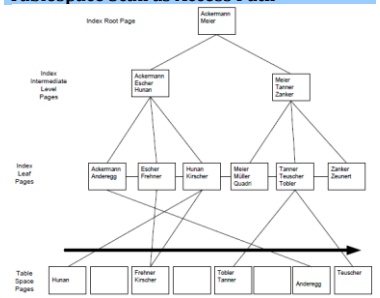
```
(Meier,Hans,RID1), (Meier,Hans,RID2),
(Meier,Hans,RID3), …
wird zu
(Meier,Hans,RID1,RID2,RID3,…)
```

Table Space with Index on Table

- File (Table Space) Einträge im allgemeinen sind nicht sortiert (im Gegensatz zu Index File Einträgen)



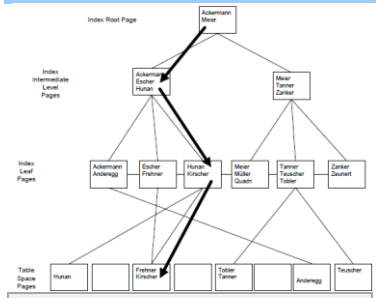
Tablespace Scan as Access Path




```
SELECE Name, Adress, …
FROM Table
WHERE Name BETWEEN 'Escher' AND 'Tobler'
```

- Wenn ein Index für den Zugriff nicht benützt wird oder werden kann, wählt das System als **Zugriffspfad (access path)** einen **Table Space Scan** (Table File Scan, Table Scan), das heisst, dass **alle** Table File Pages gelesen werden, was sehr aufwendig sein kann

An Access Path via Index Access




```
SELECT Name, Address, …
```

```
FROM Table
WHERE Name = 'Kirscher'
```

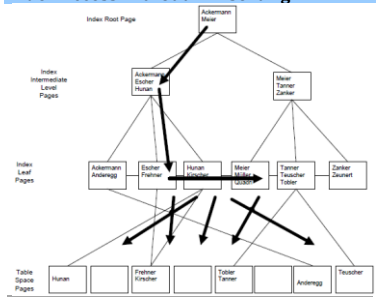
- Beim Zugriff via Index beginnt das System mit der Root Page und hangelt sich den Index Levels herunter zur Leaf Page, die den Eintrag für 'Kirscher' enthält, anschliessend, falls 'Kirscher' vorhanden, noch auf die Table Page, auf die der Indexeintrag zeigt
- Hat der Index 3 Levels, und kommt 'Kirscher' höchstens einmal vor, dann ist das Ergebnis da nach höchstens vier Page Zugriffen (**getpage**)
- 'Kirscher' kommt sicher dann höchstens einmal vor, wenn der Index als UNIQUE definiert ist

Wie kommt der Zugriffs Kosten Vergleich zustande?

- System versucht in beiden Fällen die Anzahl Page Zugriffe (getpage) zu schätzen, und vergleicht dann, nimmt den billigeren Pfad
- Kosteneinheit ist also ein Getpage
- Damit Schätzung realistisch, muss Admin von Zeit zu Zeit einen Statistik Lauf machen (Utility, System Hilfsprogramm)
  - dadurch geraten viele Grössen in die Systemtabellen

P(T1)	Anzahl Pages, auf denen Zeilen der Tabelle T1 liegen
D(I1)	Anzahl Levels des (oben) betrachteten Index i1
L(XpersStr)	Anzahl Leaf Pages des Index XpersStr
W(Table.Name)	Anzahl verschiedener (Namen-) Werte, die in der Tabelle vorkommen
FF(Name='Müller')	Filterfaktor für das Prädikat Name='Müller'
P(Table)	Anzahl Pages
Z(Table)	Anzahl Zeilen

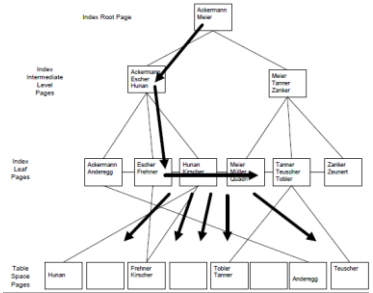
Index Access without RID sorting




```
SELECT Name, Adress, …
FROM Table
WHERE Name BETWEEN 'Frehner' AND 'Meier'
ORDER BY Name
```

- Zugriffspfad sticht von Rootpage herunter zum ersten Leafpage Eintrag 'Frehner', von da weg den Leafpages entlang bis zum letzten 'Meier'
- Für jeden Index Leafpage Eintrag wird der zugehörige Table File Eintrag geholt (via RID). So ist der Output zwar sortiert, aber viele Tablespace Pages müssen eventuell mehrmals gelesen werden

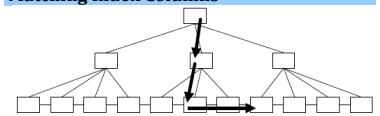
Index Access with RID sorting




```
SELECT Name, Adress, …
FROM Table
WHERE Name BETWEEN 'Frehner' AND 'Meier'
```

- **kein** sortierter Output verlangt
- gesuchte RIDs können aus den Index Leafpages gesammelt und **anschliessend sortiert** werden
- mit sortierter RID Liste muss jede Tablepage höchstens einmal gelesen werden (in aufsteigender Reihenfolge)

Matching Index Columns



- Optimizer versucht, den **Sektor der besuchten Leaf Pages** möglichst klein zu halten

Matchcols

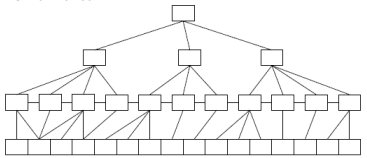
- Maximale Anzahl Index Spalten, die gebraucht werden können zur Beschränkung des zu besuchenden Sektors von Index Leafpages heisst Number of Matching Index Columns (matchcols)

Clustered and Unclustered Indexes

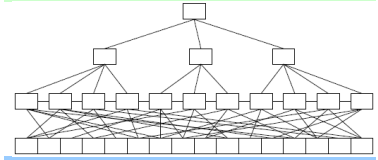
- **Clusterratio** misst den Grad der "clusteredness" (Clusterratio nahe 1 vermeidet die Notwendigkeit, Table Pages mehrmals zu lesen)

Clustered Index

- wenn a1 < a2 dann RID1 < RID2

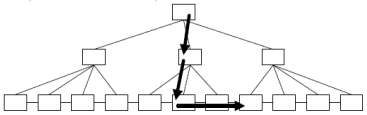


Unclustered Index



Kostenberechnung von Zugriffspaths

- betrachte WHERE Name='Müller' AND City='Luzern' und den Index(Name, Vorname, Stadt)



- es muss auf D(Index)-1 Nonleaf Pages und die Pages des zu besuchenden Sektors von Leafpages zugegriffen werden. Der Sektor definiert sich durch Name='Müller' (Matchcols=1)
- Annahme: Alle Namen kommen gleich oft vor (gleichverteilt)
  → Sektor hat ungefähr $\left(\frac{1}{W(Table.Name)}\right) * L(Index)$ Pages
  - Der Anteil FF(Name='Müller') = (1/W(Table.Name)) von Einträgen mit Name='Müller' ist der Filterfaktor für das Prädikat Name='Müller'

Tablespace Access from Index

- betrachte: WHERE Name='Müller' AND Stadt='Luzern' und den Index(Name, Vorname, Stadt)
- Während dem Lesen des Sektors für Name='Müller' pickt das System die RIDs aller Einträge heraus, die auch noch die Bedingung Stadt='Luzern' erfüllen
  - Deren Anzahl ist geschätzt
    FF(Name='Müller' AND City='Luzern') * Z(Table)
    = FF(Name='Müller') * FF(City='Luzern) * Z(Table) = N
- FF(x AND y) = FF(x) * FF(y) reflektiert die **Annahme, dass die beiden Bedingungen voneinander unabhängig seien**
- Im allgemeinen Fall ergibt die lineare Interpolation
  FF(Name='Müller' AND City='Luzern') * (y*P(Table)+(1-y)*Z(Table)) Tablespace Page Getpages

Tablespace Access from Index with RID Sort

- betrachte: WHERE Name='Müller' AND Stadt='Luzern' und den Index(Name, Vorname, Stadt) und SELECT List verlangt Tablespace Access
- Wenn die N = FF(Name='Müller' AND City='Luzern')*Z(Table) RIDs gebraucht werden für Tablespace Access, sortiert werden vor dem Tablespace Access, dann ist die **Formel von Cardenas**

$$\left(1-\left(1-\frac{1}{P}\right)^{N}\right) * P$$

die bessere Schätzung für die Anzahl Tablespace Pages, welche Einträge enthalten, die zu diesen RIDs gehören
- Die Kosten für das Sortieren der N RIDs sind dann noch

$$0.001 * N * log2(N)$$

(0.001 ist der CPU-I/O recalculation factor)

Gesamtkosten

- betrachte: WHERE Name='Müller' AND Stadt='Luzern' und den Index(Name, Vorname, Stadt) und SELECT List verlangt Tablespace Access

Zugriffskosten ohne RID Sort

$$(D(Index)-1)+\left(\frac{1}{W(Table.Name)}\right) * L(Index) + FF(Name =' Müller' AND\ City =' Luzern') * (y * P(Table) + (1-y) * Z(Table))$$

Zugriffskosten mit RID Sort

$$(D(Index)-1)+\left(\frac{1}{W(Table.Name)}\right) * L(Index) + 0.001 * log2(N) + \left(1-\left(1-\frac{1}{P(Table)}\right)^{N}\right) * P(Table)$$

(N = FF(Name='Müller' AND City='Luzern')*Z(Table)