موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد

درس و کنکور ارشد

پایگاه داده‌ها

(شاخص‌گذاری Indexing)

ویژه‌ی داوطلبان کنکور کارشناسی ارشد مهندسی کامپیوتر و IT

براساس کتب مرجع

راما کریشنان، آبراهام سیلبرشاتز و رامز المصری

ارسطو خلیلی‌فر

۱۰ شاخص‌گذاری Indexing

مقدمه

یک پایگاه داده عملیاتی با جداول بدون شاخص، تنها زمانی که تعداد کمی رکورد در جداول وجود داشته باشد، سرعت قابل قبولی در اجرای پرس و جوها خواهد داشت، اما با افزایش تدریجی رکوردها، عملا با کندی سرعت جستجو و بازیابی اطلاعات مواجه خواهد شد. ممکن است شما هم این مورد را تجربه کرده باشید که در زمان توسعه‌ی یک برنامه کاربردی، کلیه‌ی عملیات جستجو و بازیابی اطلاعات با سرعت خوبی اجرا می‌شود، اما بعد از تحویل به مشتری و گذشت یک مدت زمان، با گله و شکایت کندی سرعت از سوی مشتری مواجه می‌شوید. در انتهای اغلب کتاب‌ها شاخص(Index) وجود دارد، به این معنا که لیست کلمات و اصطلاحات مهم(کلیدها) کتاب به ترتیب حروف الفبا، به همراه شماره صفحاتی که آن کلمات استفاده شده (آدرس رکوردها)، آورده می‌شود. مهمترین کاربرد شاخص، افزایش سرعت جستجو و بازیابی اطلاعات است. به نحوی که در صورت عدم وجود فهرست مطالب در یک کتاب، می بایست تک تک صفحات کتاب را جهت یافتن مطلب مورد نظر، برگ می‌زدیم. پس شاخص‌ها بر مبنای کلیدهای جستجو و آدرس رکوردها ساخته می‌شوند. شاخص باعث بالا رفتن سرعت دستیابی به اطلاعات می‌گردد. تکنیک شاخص‌بندی یا شاخص‌گذاری (Indexing) در اکثر نرم‌افزارهای امروزی استفاده می‌شود. شاخص‌ها برای بهبود فرآیند جستجو و بازیابی اطلاعات در جداول ایجاد می‌شوند. به عبارت دیگر شاخص‌ها به فرآیند جستجو و بازیابی اطلاعات، سرعت می‌بخشند. شاخص‌ها باعث می‌شوند موتور جستجوی پایگاه داده کل یک جدول را برای پیداکردن رکورد یا رکوردهای مورد نظر به طور کامل نگردد. اساسا شاخص‌ها بر روی ستون‌هایی باید تنظیم شود که بیشتر مورد جستجو قرار می‌گیرند.

دو هدف اصلی سیستم ذخیره و بازیابی اطلاعات در پایگاه داده‌ها، اول سرعت عملیات در ذخیره و بازیابی اطلاعات و دوم صرفه‌جویی در مصرف حافظه است. برای مثال کاهش افزونگی

محتوایی (طبیعی) توسط نرمال‌سازی جداول منجر به کاهش میزان حافظه مصرفی می‌شود. عمل واکشی تک تک رکوردها وقت‌گیر است، برای رفع این عیب، شاخص یا Index ابداع شد. برای اینکه جستجو و بازیابی داده‌ها با سرعت و کارایی بیشتر صورت گیرد، از شاخص استفاده می‌شود. شاخص ساختمان داده‌ای است که سیستم مدیریت پایگاه داده‌ها به کمک آن رکوردهای مورد نظر را در یک فایل با سرعت بسیار زیاد پیدا می‌کند و به این ترتیب سرعت پاسخ به پرس و جوها افزایش می‌یابد. هر ساختار شاخص، حاوی رکوردهایی است که در هر رکورد یک مقدار کلیدی (کلید جستجو) و یک اشاره‌گر شامل آدرس فیزیکی رکوردهای جدول پایه در آن نگهداری می‌شود. مطابق شکل زیر:

Search Key	Pointer

اغلب سیستم‌های مدیریت پایگاه داده‌ها، از ساختار درخت برای ایجاد شاخص‌ها استفاده می‌کنند. عمق درخت، بیشترین تعداد سطوح از ریشه به برگ است. عمق ممکن است در مسیرهای مختلف از ریشه تا برگ متفاوت باشد. و همچنین عمق ممکن است در مسیرهای مختلف از ریشه تا برگ یکسان باشد، که در این شرایط با درخت متوازن و B^+-Tree مواجه هستیم. هرچه درجه‌ی گره‌های درخت بیشتر شود، درخت پهن‌تر و کم‌عمق‌تری ایجاد می‌شود. از آنجاییکه زمان دسترسی در یک درخت، بیشتر وابسته به عمق درخت است تا پهنای آن، پس ساخت درخت پهن و کم‌عمق در ایجاد شاخص باعث افزایش سرعت جستجو می‌شود، ساختارهای B^+-Tree درخت‌هایی با عمق کم و پهنای زیاد هستند. ساختار B^+ Tree index برای پاسخ به Range Query و Equality Query مناسب است. بنابراین اعمال سیاست شاخص‌گذاری توسط ساختار B^+ Tree باعث می‌شود Range Queryها و Equality Queryهای مرتبط با ستون مورد نظر، با سرعت بیشتری انجام شود.

توجه: یک شاخص روی مقادیر یک یا چند ستون از جدول تعریف می‌شود. و به ستون‌هایی که شاخص روی آنها تعریف می‌شود، **کلید جستجو**(Search Key) گفته می‌شود. در واقع در شاخص به جای نگهداری کلیه اطلاعات یک جدول، فقط مقادیر کلید جستجو نگهداری می‌شود.

توجه: اغلب شاخص روی کلید اصلی تعریف می‌شود، که در این حالت کلید جستجو همان کلید اصلی خواهد بود. اما هیچ الزامی برای این امر وجود ندارد و می‌توان با تعریف شاخص روی هر یک از ستون‌ها آنرا به عنوان کلید جستجو در نظر گرفت. بنابراین امکان تعریف بیش از یک شاخص روی یک جدول وجود دارد.

توجه: امکان تعریف یک شاخص روی چندین ستون به صورت ترکیبی نیز وجود دارد، در اینصورت کلید جستجو شامل چندین ستون است که به آن کلید جستجوی مرکب (Composite

Search Key) می‌گویند. در این حالت مبنای مرتب‌سازی رکوردها ترکیب ستون‌هاست. برای مثال اگر شاخص روی ترکیب ستون‌های (X,Y) تعریف شده باشد، آنگاه $(x_1,y_1)<(x_2,y_2)$ است، اگر $x_1<x_2$ و یا $x_1=x_2$ و $y_1<y_2$

توجه: اگر شاخص روی یک ستون یا چندستون(ترکیبی) تعریف شده باشد، فقط سرعت جستجوهای مبتنی بر آن ستون یا چندستون را افزایش می‌دهد.

توجه: تعریف و نگهداری شاخص موجب تحمیل سربار حافظه‌ای به سیستم می‌شود. شاخص بر روی هارد دیسک نگهداری می‌شود و هنگام استفاده به حافظه اصلی آورده می‌شود، بنابراین اعمال سیاست شاخص‌گذاری، بر افزایش حجم اطلاعات ذخیره شده بر روی حافظه اصلی و هارد دیسک تاثیر دارد.

توجه: عملیات درج، حذف و بروزرسانی در جدولی که شاخص دارد یعنی خود جدول پایه، نسبت به جدولی که شاخص ندارد زمان بیشتری مصرف می‌کند و به تبع این عملیات کندتر خواهد بود. زیرا شاخص‌ها نیز همگام با جداول پایه خود نیاز به بروزرسانی دارند. بنابراین تنها روی ستون‌هایی شاخص ایجاد می‌گردد، که به تناوب روی آنها جستجو انجام می‌شود. بنابراین هرچه تعداد شاخص‌های یک جدول بیشتر باشد، سرعت Insert، Update و Delete در آن جدول کمتر می‌شود. اما خود شاخص عامل جستجو و بازیابی سریع اطلاعات از یک جدول پایه است.

توجه: شاخص‌گذاری، افزونگی تکنیکی دارد و مقداری از فضای حافظه اصلی و هارد دیسک را اشغال می‌کند. به عبارت دیگر تعریف هر شاخص روی یک جدول هزینه‌ی زیادی را به پایگاه داده تحمیل می‌کند و اگر نرخ تغییرات محتوایی پایگاه داده زیاد باشد این هزینه به صورت قابل توجهی افزایش می‌یابد. بنابراین طراحان پایگاه داده ترجیح می‌دهند که روی هر جدول بیش از یک شاخص تعریف نکنند.

توجه: شاخص‌گذاری، منجر به افزایش سرعت جستجو و بازیابی اطلاعات و به تبع کاهش زمان جستجو و بازیابی اطلاعات می‌شود، نبود شاخص باعث کندی سرعت جستجو و زیادی آن باعث کندی فرآیندهای درج، حذف بروزرسانی رکوردها در جداول پایه می‌شود. بنابراین راز نه در افراط است و نه در تفریط بلکه راز در تعادل است، گاهی هم استفاده‌ی مکرر از گزینه‌های خوب، نتیجه‌ی بد هم می‌تواند به همراه داشته باشد! و به قول معروف در حوزه‌ی سلامتی نیاکان ما گفته‌اند که کم بخور، همیشه بخور...

توجه: شاخص (Index) با هدف افزایش سرعت جستجو و بازیابی اطلاعات و به تبع کاهش زمان جستجو و بازیابی اطلاعات ایجاد می‌گردد. و به عنوان نمونه‌ی دیگری از فراداده‌ها، مشخصات سیستمی شاخص در کاتالوگ سیستم نگهداری می‌شود.

پیاده‌سازی مفهوم شاخص در SQL-DDL

برای **ایجاد** یک شاخص روی یک جدول پایه، از دستور CREATE INDEX استفاده می‌شود. با اجرای این دستور، تعریف مشخصات شاخص در کاتالوگ نیز درج می‌شود.

فرم کلی ایجاد شاخص به صورت زیر است:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name
ON table_name (column1 [ASC | DESC], column2 [ASC | DESC], ...);
```

برای **حذف** یک شاخص تعریف شده روی یک جدول پایه، از دستور DROP INDEX استفاده می‌شود. با اجرای این دستور، تعریف مشخصات شاخص از کاتالوگ نیز حذف می‌شود.

فرم کلی حذف شاخص به صورت زیر است:

```
DROP INDEX index_name
```

به طور کلی دو نوع شاخص وجود دارد:

(۱)شاخص از نوع مرتب‌شده و (۲) شاخص از نوع Hash که در ادامه به بررسی آن می‌پردازیم.

۱- شاخص از نوع مرتب‌شده(Ordered Index)

همانطور که گفتیم، یک شاخص روی مقادیر یک یا چند ستون از جدول تعریف می‌شود. و به ستون‌هایی که شاخص روی آنها تعریف می‌شود، **کلید جستجو**(Search Key) گفته می‌شود. در واقع در شاخص به جای نگهداری کلیه اطلاعات یک جدول پایه، فقط مقادیر کلید جستجو نگهداری می‌شود. جهت جستجوی سریع‌تر، در شاخص مرتب‌شده مقادیر کلید جستجو به صورت مرتب‌شده نگهداری می‌شود، که علت نامگذاری نیز همین بوده است.

توجه: شاخص مرتب‌شده به کمک دو ساختار آرایه و ساختار B^+ Tree قابل پیاده‌سازی است.

توجه: در SQL Server شاخص مرتب‌شده به کمک ساختار B^+ Tree پیاده‌سازی شده است. به عبارت دیگر مقادیر شاخص در ساختار B^+ Tree نگهداری می‌شود.

توجه: به طور کلی، درخت یک ساختمان داده غیرخطی است که برای عملیاتی مانند **جستجو** مورد استفاده قرار می‌گیرد.

توجه: انجام عملیات در B^+ Tree، نسبت به سایر درخت‌ها سریعتر است، بنابراین در SQL Server ساختار B^+ Tree جهت نگهداری مقادیر شاخص مورد استفاده قرار می‌گیرد.

توجه: شاخص مرتب‌شده به دو صورت clustered Index و Nonclustered Index وجود دارد.

شاخص مرتب‌شده به صورت clustered Index

توجه: هر جدول می‌تواند حداکثر **یک** clustered Index داشته باشد.

توجه: زمانی که کلید اصلی یک جدول در SQL Server تعریف می شود اولین شاخص بر روی

آن جدول ایجاد می‌شود که به آن clustered Index گفته می‌شود.

توجه: در clustered Index ترتیب منطقی رکوردها در جدول پایه با ترتیب فیزیکی چیدمان آن‌ها بر روی هارد دیسک یکسان است. بنابراین هر جدول می‌تواند حداکثر یک clustered Index داشته باشد. حفظ ترتیب منطقی رکوردها در جدول پایه با ترتیب فیزیکی چیدمان آن‌ها بر روی هارد دیسک توسط بخشی بنام Row Offset Array و توسط اشاره‌گرها انجام می‌گردد.

توجه: به شاخص مرتب‌شده به صورت clustered Index، شاخص اصلی یا اولیه نیز می‌گویند.

توجه: مقادیر شاخص مرتب‌شده به صورت clustered Index باید یکتا و منحصر به فرد باشد و UNIQUE تعریف گردد.

توجه: شاخص مرتب‌شده به صورت clustered Index، منجر به این می‌شود که رکوردهای یک جدول پایه به صورت ترتیبی و **صعودی** بر روی هارد دیسک ذخیره شوند.

قطعه کد ایجاد clustered Index به صورت زیر است:

```
CREATE UNIQUE CLUSTERED INDEX index_name
ON table_name (column1 ASC, column2 ASC, ...);
```

توجه: زمانی که کلید اصلی یک جدول در SQL Server تعریف می‌شود قطعه کد فوق به طور خودکار اجرا می‌شود.

توجه: به طور پیش فرض و غیرقابل تغییر در ایجاد clustered Index دستور UNIQUE استفاده می‌گردد، تا یکتایی و منحصر به فرد بودن مقادیر شاخص **گارانتی** گردد.

توجه: به طور پیش فرض و غیرقابل تغییر در ایجاد clustered Index دستور Ascending یا ASC استفاده می‌گردد، تا **صعودی** بودن مقادیر شاخص **گارانتی** گردد.

مثال: با اجرای قطعه کد زیر جدول S با کلید اصلی S# و به تبع یک clustered Index روی ستون S# ایجاد می‌گردد.

```
CREATE TABLE S (
    S# int NOT NULL,
    SName varchar(255),
    City varchar(255),
    PRIMARY KEY (S#)
);
```

قطعه کد ایجاد clustered Index که بطور خودکار اجرا می‌شود، به صورت زیر است:

```
CREATE UNIQUE CLUSTERED INDEX PK_S
ON S (S# ASC);
```

جداول زیر را در نظر بگیرید:

S#	Sname	City
S1	Sn1	C1
S2	Sn2	C2
S3	Sn3	C2
S4	Sn4	C3
S5	Sn5	C4
S6	Sn6	C5
S7	Sn7	C6
S8	Sn8	C7
S9	Sn9	C7

جدول S

S#	P#	QTY
S1	P1	10
S1	P2	20
S2	P1	30

جدول SP

P#	Pname	Color
P1	Pn1	Red
P2	Pn2	Blue

جدول P

با استفاده از clustered Index پیش فرض و موجود روی ستون #S، سرعت پاسخگویی به تمامی پرس و جوهایی که جستجو را بر اساس #S در جدول S انجام می‌دهند، افزایش می‌یابد.

مثال:

```
SELECT S#
FROM S
WHERE S#='S9'
```

اما در مورد پرس و جوهایی که جستجو را بر اساس ستون #S انجام نمی‌دهند. تعریف شاخص فوق هیچ تاثیری در سرعت پاسخگویی به پرس و جو ندارد.

توجه: در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه به دلیل مزایای ساختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.

ساختار B^+ Tree

B^+ Tree به معنای درخت متوازن (Balanced Tree) است و نه درخت دودویی(Binary Tree) و یا درخت جستجوی دودویی(BST-Binary Search Tree).

توجه: ساختار B^+ – Tree یک درخت جستجو است. در این ساختار سه نوع گره ریشه(Root Node)، گره میانی(Intermediate Node) و گره برگ(Leaf Node) وجود دارد که همه برگ‌هایش در یک سطح هستند. به عبارت دیگر ساختار B^+ – Tree از سطوح ریشه، میانی و برگ تشکیل شده است.

توجه: با توجه به متوازن بودن ساختار B^+ – Tree و فاصله یکسان همه گره‌های موجود در سطح برگ تا ریشه، جهت دسترسی به هر یک از گره‌های موجود در سطح برگ مراحل یکسانی لازم است. به عبارت دیگر تمامی گره‌های موجود در سطح برگ(leaf) دارای فاصله(عمق) یکسانی تا

ریشه هستند. به دلیل اینکه در هر عملیاتی تعداد مراجعه به دیسک متناسب با تعداد سطوح درخت است. بنابراین زمانی که همه گره‌های سطح برگ دارای عمق یکسانی باشند، انجام عملیات در هر گره هزینه یکسانی با سایر گره‌ها دارد.

توجه: به دلیل وجود ارتباط(اشاره‌گر) بین گره‌ها در این ساختار، پیمایش در B^+ Tree به سرعت انجام می شود.

توجه: اصطلاحا $B^+ - \text{Tree}$ را درجه t می‌گویند $(t \geq 2)$ که داخل هر گره‌اش حداقل $\left\lceil \left(\frac{t}{2}\right) \right\rceil - 1$ و حداکثر $t-1$ کلید جستجو است. همچنین تعداد فرزندان(اشاره‌گرها) یکی از تعداد کلیدها بیشتر است، بنابراین هر گره غیر برگ حداقل $\left\lceil \left(\frac{t}{2}\right) \right\rceil$ فرزند و حداکثر t فرزند دارد. البته ریشه استثنا است و می‌تواند حداقل شامل یک کلید جستجو باشد، یعنی حداقل دو فرزند داشته باشد، اما ریشه حداکثر همان $t-1$ کلید جستجو را می‌تواند داشته باشد، یعنی حداکثر t فرزند. در یک قاعده‌ی کلی گره‌ای که xتا کلید داشته باشد، اگر برگ نباشد، دقیقا $x+1$ فرزند دارد.

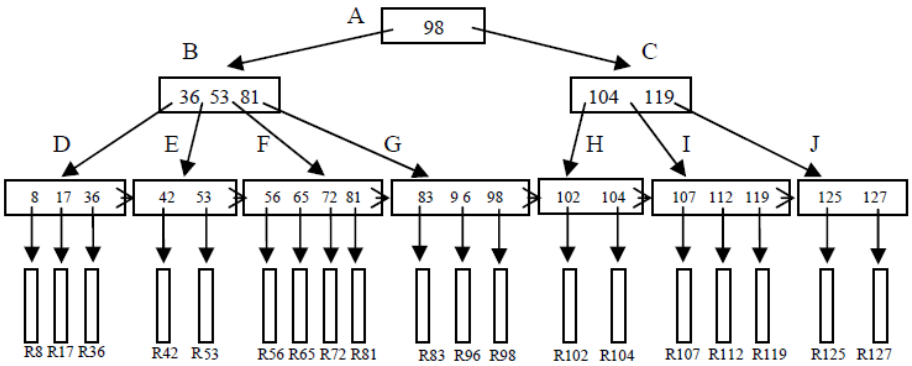
نتیجه: $\left\lceil \left(\frac{t}{2}\right) \right\rceil - 1 \leq$ **تعداد کلید جستجو در** $B^+ - \text{Tree}$ **درجه t** $\leq t-1$

نتیجه: $\left\lceil \left(\frac{t}{2}\right) \right\rceil \leq$ **تعداد فرزند(اشاره‌گر) در** $B^+ - \text{Tree}$ **درجه t** $\leq t$

توجه: در ساختار $B^+ - \text{Tree}$ اشاره‌گرهای موجود در ریشه و گره‌های میانی به گره‌های فرزند اشاره می‌کنند، اما اشاره‌گرهای موجود در برگ‌ها به سطرها یا رکوردهای مربوطه در دیسک اشاره می‌کنند.

توجه: در B^+ Tree تمام کلیدهای جدول پایه در برگ‌ها نگهداری شده و برگ‌ها به یکدیگر متصل می‌شوند تا یک مسیر سریالی و ترتیبی برای پیمایش کلیدها در درخت را فراهم سازد.

توجه: استفاده از B^+ Tree سرعت جستجو و بازیابی اطلاعات مورد نظر را افزایش می‌دهد. برای جستجو و بازیابی اطلاعات مورد نظر با استفاده از B^+ Tree عملیات جستجو از ریشه آغاز می‌شود و تا رسیدن به یکی از برگ‌ها ادامه پیدا می‌کند. برای مثال شکل زیر یک B^+ Tree است:



در شکل فوق برای جستجو و بازیابی(lookup) رکورد وابسته به کلید 53 یعنی R53 ابتدا کلید 53 با 98 یعنی ریشه مقایسه می‌شود، چون کمتر از آن است کنترل به سراغ گره B می‌رود. کلید 53، از 36 بزرگتر و از 53 کوچکتر یا مساوی است لذا به سراغ گره E می‌رود. در ساختار B^+ Tree فقط کلیدهای موجود در برگ‌ها حاوی اشاره‌گرهایی به رکوردهای جدول پایه هستند. لذا عمل جستجو وقتی تمام می‌شود که کلید مورد جستجو در برگ‌ها پیدا شود. به لیست پیوندی برگ‌ها مجموعه توالی یا مجموعه دنباله می‌گویند.

توجه: در ساختار B^+-Tree به دلیل وجود چندین کلید جستجو در هر گره، رشد تعداد سطوح به کندی انجام می‌شود. به عنوان مثال جهت شاخص‌گذاری روی اطلاعات چندین میلیون رکورد از یک جدول پایه و درج و نگهداری مقادیر شاخص در ساختار B^+-Tree، سطح میانی تنها دارای دو یا سه سطح است و به ندرت یک B^+-Tree با شش سطح میانی ایجاد می‌شود. در واقع با توجه به وابستگی هزینه عملیات جستجو به تعداد سطوح، کم بودن تعداد سطوح از نقاط قوت ساختار B^+-Tree محسوب می‌شود.

عمل درج در ساختار B^+ Tree

مثال: درج رشته زیر را در نظر بگیرید:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

توجه: فرض کنید درجه t=5 و به تبع حداکثر تعداد کلید جستجو در هر گره t-1 یعنی 5-1=4 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	4	
Number of Pointers	5	

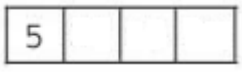
قوانین درج در درخت B^+ Tree **به صورت زیر است:**

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node
YES	NO	**leaf node** **Split into two nodes** 1- keys < middle key go to the left leaf node 2- keys >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node
NO	YES	**Index node** **Split into two nodes** 1- key < middle key go to the left index node 2- key >= middle key go to the right leaf node 3-**move** the smallest key of right leaf node to the parent node

توجه: اگر تعداد مجموعه کلیدهای جستجو **فرد** بود آنگاه عدد وسط برابر middle key خواهد بود. اما اگر تعداد مجموعه کلیدهای جستجو **زوج** بود آنگاه کلیدهای جستجو به طور مساوی در در دو گره سمت چپ و راست قرار می‌گیرند. و middle key کوچکترین عضو مجموعه گره سمت راست خواهد بود.

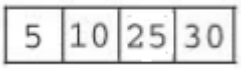
ابتدا رکورد اول با کلید 5 درج می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95



سپس رکوردهای بعدی با کلیدهای 10، 25 و 30 به ترتیب اما به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95



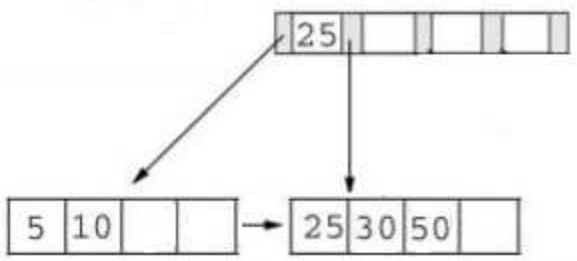
در ادامه رکورد بعدی با کلید 50 درج می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 50 چون گره مورد نظر پر است (5,10,25,30,50) عمل تقسیم گره صورت می‌گیرد:

Leaf node Full	Index node Full	**Action**
YES	NO	**leaf node** **Split into two nodes** 1- keys < middle key go to the left leaf node 2- keys >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node

left leaf node	**right leaf node**
(5,10)	(25,30,50)
leaf node **Split into two nodes** 1- keys < middle key go to the left leaf node 2- key >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node **توجه:** کوچکترین عدد در right leaf node یعنی مجموعه اعداد (50 ,30 ,**25**) یعنی عدد 25 به گره بالاتر فرستاده (**Copy**) می‌شود.	
(**25**)	
(5,10)	(**25**,30,50)



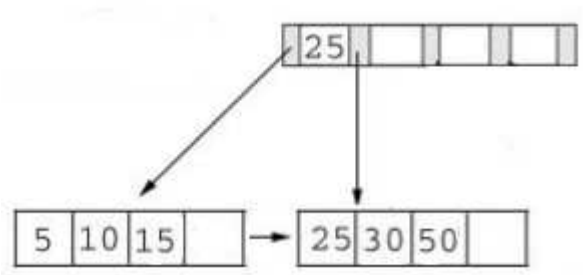
در ادامه رکورد بعدی با کلید 15 به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 15 چون گره مورد نظر پر نیست (5,10,15) عمل تقسیم گره صورت نمی‌گیرد.

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node

left leaf node	**right leaf node**
(5,10,15)	(25,30,50)
Place the record in sorted position in the appropriate leaf node **توجه:** فقط رکورد بعدی با کلید 15 به صورت مرتب شده درج و جایگذاری می‌شود.	



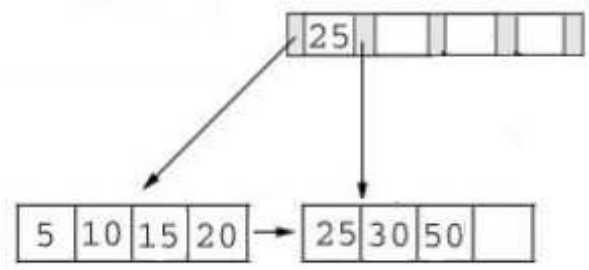
در ادامه رکورد بعدی با کلید 20 به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 20 چون گره مورد نظر پر نیست (5,10,15,20) عمل تقسیم گره صورت نمی‌گیرد.

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node

left leaf node	**right leaf node**
(5,10,15,20)	(25,30,50)
Place the record in sorted position in the appropriate leaf node **توجه:** فقط رکورد بعدی با کلید 20 به صورت مرتب شده درج و جایگذاری می‌شود.	



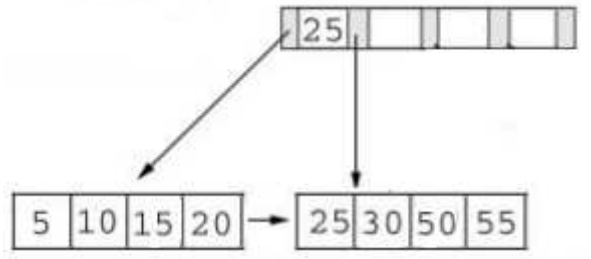
در ادامه رکورد بعدی با کلید 55 به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 55 چون گره مورد نظر پر نیست (25,30,50,55) عمل تقسیم گره صورت نمی‌گیرد.

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node

left leaf node	**right leaf node**
(5,10,15,20)	(25,30,50,55)
Place the record in sorted position in the appropriate leaf node **توجه:** فقط رکورد بعدی با کلید 55 به صورت مرتب شده درج و جایگذاری می‌شود.	



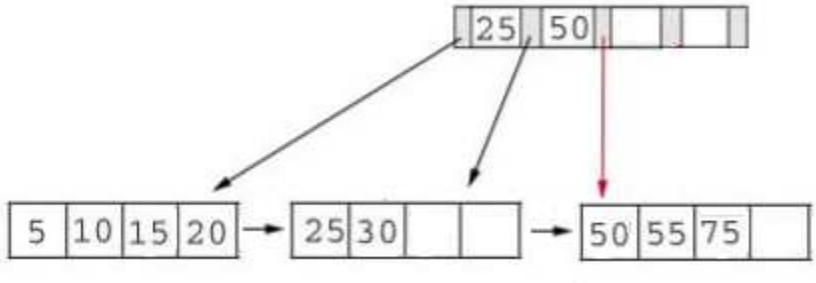
در ادامه رکورد بعدی با کلید 75 درج می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 75 چون گره مورد نظر پر است (25,30,50,55,75) عمل تقسیم گره صورت می‌گیرد:

Leaf node Full	Index node Full	**Action**
YES	NO	**leaf node** **Split into two nodes** 1- keys < middle key go to the left leaf node 2- keys >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node

left leaf node	**right leaf node**
(25,30)	(50,55,75)
leaf node **Split into two nodes** 1- keys < middle key go to the left leaf node 2- key >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node **توجه:** کوچکترین عدد در right leaf node یعنی مجموعه اعداد (**50**,55,75) یعنی عدد 50 به گره بالاتر فرستاده (**Copy**) می‌شود.	
(**50**)	
(25,30)	(**50**,55,75)



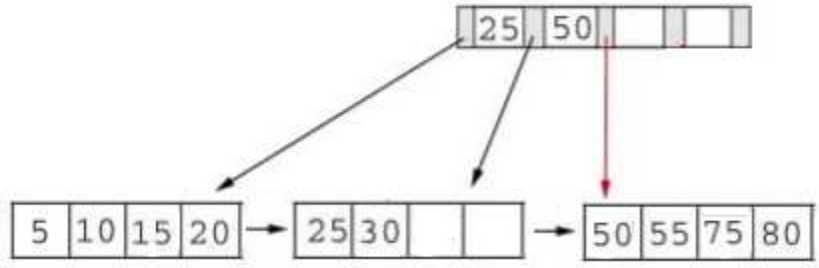
در ادامه رکورد بعدی با کلید 80 به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 80 چون گره مورد نظر پر نیست (50,55,75,80) عمل تقسیم گره صورت نمی‌گیرد.

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node

left leaf node	**right leaf node**
(25,30)	(50,55,75,80)
Place the record in sorted position in the appropriate leaf node **توجه:** فقط رکورد بعدی با کلید 80 به صورت مرتب شده درج و جایگذاری می‌شود.	



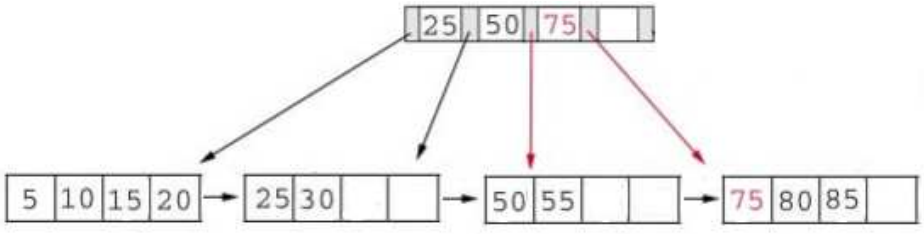
در ادامه رکورد بعدی با کلید 85 درج می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 85 چون گره مورد نظر پر است (50,55,75,80,85) عمل تقسیم گره صورت می‌گیرد:

Leaf node Full	Index node Full	**Action**
YES	NO	**leaf node** **Split into two nodes** 1- keys < middle key go to the left leaf node 2- keys >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node

left leaf node	**right leaf node**
(50,55)	(75,80,85)
leaf node **Split into two nodes** 1- keys < middle key go to the left leaf node 2- key >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node **توجه:** کوچکترین عدد در right leaf node یعنی مجموعه اعداد (**75**,80,85) یعنی عدد 75 به گره بالاتر فرستاده (**Copy**) می‌شود.	
(**75**)	
(50,55)	(**75**,80,85)



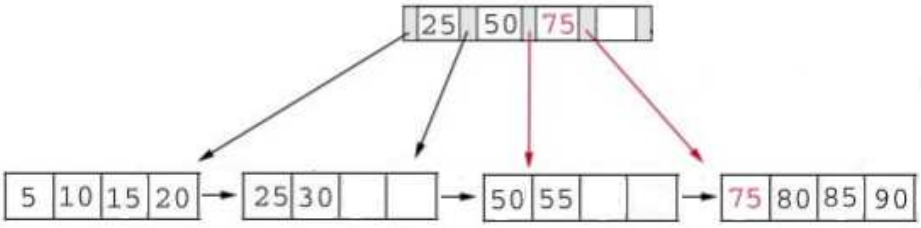
در ادامه رکورد بعدی با کلید 90 به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 90 چون گره مورد نظر پر نیست (75,80,85,90) عمل تقسیم گره صورت نمی‌گیرد.

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node

left leaf node	**right leaf node**
(50,55)	(75,80,85,90)
Place the record in sorted position in the appropriate leaf node **توجه:** فقط رکورد بعدی با کلید 90 به صورت مرتب شده درج و جایگذاری می‌شود.	



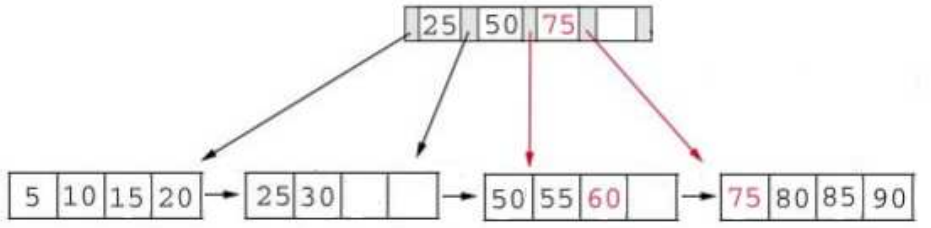
در ادامه رکورد بعدی با کلید 60 به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 60 چون گره مورد نظر پر نیست (50,55,60) عمل تقسیم گره صورت نمی‌گیرد.

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node

left leaf node	**right leaf node**
(50,55,60)	(75,80,85,90)
Place the record in sorted position in the appropriate leaf node **توجه:** فقط رکورد بعدی با کلید 60 به صورت مرتب شده درج و جایگذاری می‌شود.	



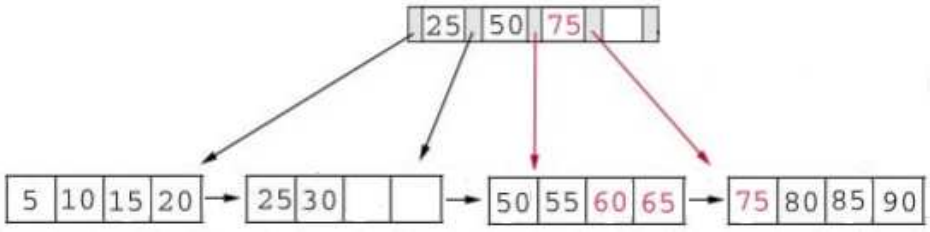
در ادامه رکورد بعدی با کلید 65 به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 65 چون گره مورد نظر پر نیست (50,55,60,65) عمل تقسیم گره صورت نمی‌گیرد.

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node

left leaf node	**right leaf node**
(50,55,60,65)	(75,80,85,90)
Place the record in sorted position in the appropriate leaf node **توجه:** فقط رکورد بعدی با کلید 65 به صورت مرتب شده درج و جایگذاری می‌شود.	



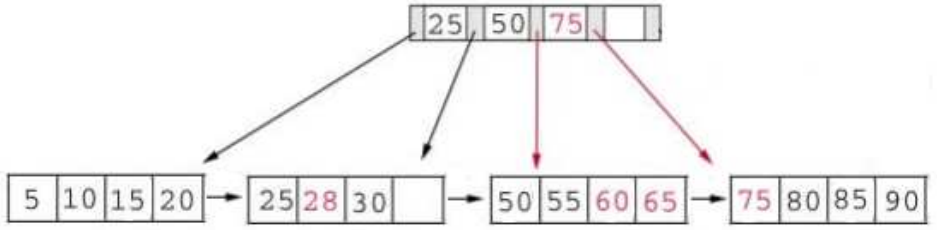
در ادامه رکورد بعدی با کلید 28 به صورت مرتب شده درج و جایگذاری می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 28 چون گره مورد نظر پر نیست (25,28,30) عمل تقسیم گره صورت نمی‌گیرد.

Leaf node Full	Index node Full	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf node

left leaf node	**right leaf node**
(5,10,15,20)	(25,28,30)
Place the record in sorted position in the appropriate leaf node **توجه:** فقط رکورد بعدی با کلید 28 به صورت مرتب شده درج و جایگذاری می‌شود.	



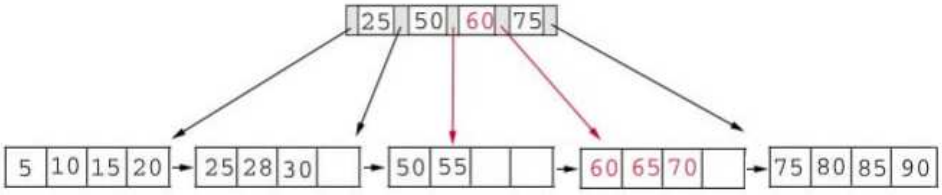
در ادامه رکورد بعدی با کلید 70 درج می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 70 چون گره مورد نظر پر است (50,55,60,65,70) عمل تقسیم گره صورت می‌گیرد:

Leaf node Full	Index node Full	**Action**
YES	NO	**leaf node** **Split into two nodes** 1- keys < middle key go to the left leaf node 2- keys >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node

left leaf node	**right leaf node**
(50,55)	(60,65,70)
leaf node **Split into two nodes** 1- keys < middle key go to the left leaf node 2- key >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node **توجه:** کوچکترین عدد در right leaf node یعنی مجموعه اعداد (**60**,65,70) یعنی عدد 60 به گره بالاتر فرستاده (**Copy**) می‌شود.	
(**60**)	
(50,55)	(**60**,65,70)



در ادامه رکورد بعدی با کلید 95 درج می‌شود:

5,10,25,30,50,15,20,55,75,80,85,90,60,65,28,70,95

با درج کلید 95 چون گره مورد نظر پر است (75,80,85,90,95) عمل تقسیم گره صورت می‌گیرد:

Leaf node Full	Index node Full	**Action**
YES	NO	**leaf node** **Split into two nodes** 1- keys < middle key go to the left leaf node 2- keys >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node

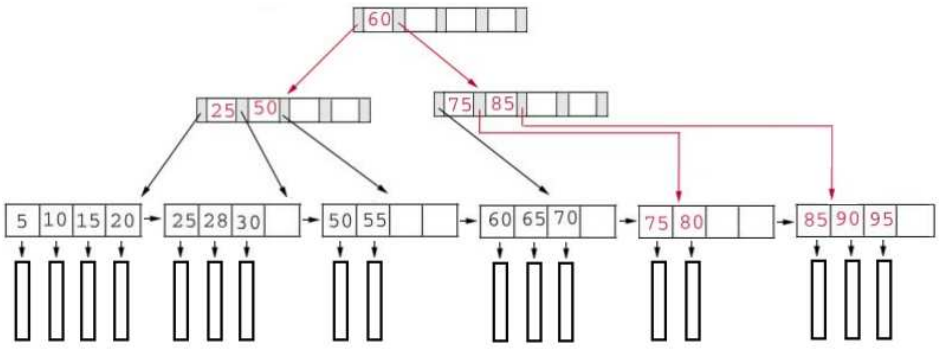
left leaf node	**right leaf node**
(75,80)	(85,90,95)
leaf node **Split into two nodes** 1- keys < middle key go to the left leaf node 2- key >= middle key go to the right leaf node 3-**copy** the smallest key of right leaf node to the parent node **توجه:** کوچکترین عدد در right leaf node یعنی مجموعه اعداد (**85**,90,95) یعنی عدد 85 به گره بالاتر فرستاده (**Copy**) می‌شود.	
(**85**)	
(75,80)	(**85**,90,95)

در ادامه کلید 85 در سطح بالاتر یعنی **ریشه** درج می‌شود.

با درج کلید 85 چون گره **ریشه** پر است (25,50,60,75,85) عمل تقسیم گره **ریشه** صورت می‌گیرد:

Leaf node Full	Index node Full	**Action**
NO	YES	**Index node** **Split into two nodes** 1- key < middle key go to the left index node 2- key >= middle key go to the right leaf node 3-**move** the smallest key of right leaf node to the parent node

left leaf node	**right leaf node**
(25,50)	(60,75,85)
Index node **Split into two nodes** 1- key < middle key go to the left index node 2- key >= middle key go to the right leaf node 3-**move** the smallest key of right leaf node to the parent node **توجه:** کوچکترین عدد در right leaf node یعنی مجموعه اعداد (**60**,75,85) یعنی عدد 60 به گره بالاتر فرستاده (**Move**) می‌شود، به کلمه‌ی **Move** و تفاوت آن با **Copy** در حالت‌های قبل دقت کنید.	
(**60**)	
(25,50)	(75,85)



به طور کلی دو نوع پرس و جو در جداول پایگاه داده انجام می‌گردد:

(۱)پرس و جوی نقطه‌ای و (۲) پرس و جوی بازه‌ای که در ادامه به بررسی آن می‌پردازیم.

۱-پرس و جوی نقطه‌ای(Point query یا Equality query)

در پرس و جوی نقطه‌ای، هدف شناسایی و بازیابی سطرهایی است که مقدار ستون مورد جستجو در **یک مقدار** مشخص قرار دارد.

با اجرای قطعه کد زیر جدول S با کلید اصلی S# و به تبع یک clustered Index روی ستون S# ایجاد می‌گردد.

```
CREATE TABLE S (
    S# int NOT NULL,
    SName varchar(255),
    City varchar(255),
    PRIMARY KEY (S#)
);
```

قطعه کد ایجاد clustered Index که بطور خودکار اجرا می‌شود، به صورت زیر است:

```
CREATE UNIQUE CLUSTERED INDEX PK_S
ON S (S# ASC);
```

جداول زیر را در نظر بگیرید:

S#	Sname	City
S1	Sn1	C1
S2	Sn2	C2
S3	Sn3	C2
S4	Sn4	C3
S5	Sn5	C4
S6	Sn6	C5
S7	Sn7	C6
S8	Sn8	C7
S9	Sn9	C7

جدول S

S#	P#	QTY
S1	P1	10
S1	P2	20
S2	P1	30

جدول SP

P#	Pname	Color
P1	Pn1	Red
P2	Pn2	Blue

جدول P

توجه: با استفاده از clustered Index پیش فرض و موجود روی ستون S#، سرعت پاسخگویی به تمامی پرس و جوهایی که جستجو را بر اساس S# در جدول S انجام می‌دهند، افزایش می‌یابد.

توجه: اما در مورد پرس و جوهایی که جستجو را بر اساس ستون S# انجام نمی‌دهند. تعریف شاخص فوق هیچ تاثیری در سرعت پاسخگویی به پرس و جو ندارد.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

S1,S2,S3,S4,S5,S6,S7,S8,S9

پرس و جوی زیر را در نظر بگیرید:

```
SELECT S#
FROM S
WHERE S#='S9'
```

توجه: شرط جلوی where یعنی S#='S9' **یک مقدار** مشخص است و این یعنی **پرس و جوی نقطه‌ای**. خروجی پرس و جوی فوق به صورت زیر است:

S#
S9

پیاده‌سازی شاخص فوق با ساختار B^+ Tree

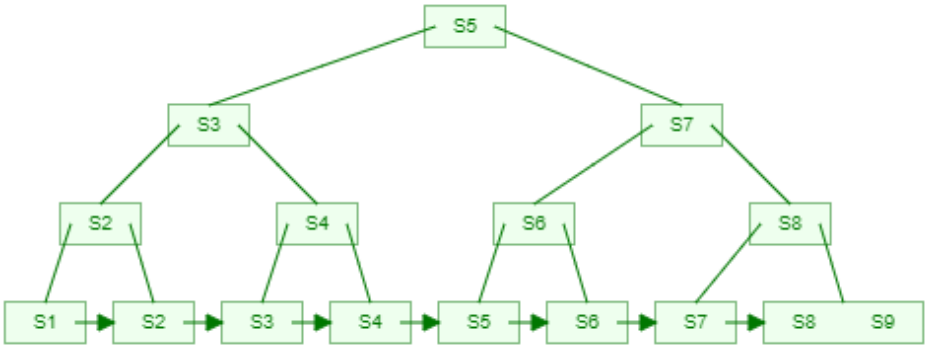
در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه بـه دلیـل مزایـای سـاختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.

توجه: فرض کنید درجه t=3 و به تبع حداکثر تعداد کلیـد جسـتجو در هـر گـره t-1 یعنـی 3-1=2 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=2	
Number of Pointers	t=3	

ساختار نهایی B^+ Tree **به صورت زیر است:**



نتیجه: رکورد S9 پس از جستجو در ساختار B^+ Tree و به صورت درختی در 4 حرکت در سطح چهارم یافت می‌شود، که مرتبه اجرایی آن $O(Log_t\ n)$ است.

توجه: به طور کلی اگر n تعداد حداکثر سطرها(رکوردها) باشد و t درجه درخت و h تعداد سطوح در ساختار $B^+ - Tree$ باشد، آنگاه روابط زیر برقرار است:

For a t-order B+ tree with h levels of index

- The minimum number of records stored is: $n_{min} = 2\left\lceil \frac{t}{2} \right\rceil^{h-1} - 2\left\lceil \frac{t}{2} \right\rceil^{h-2}$
- The maximum number of records stored is: $n_{max} = t^h - t^{h-1}$
- The minimum number of keys is: $n_{key-min} = 2\left\lceil \frac{t}{2} \right\rceil^{h-1} - 1$
- The maximum number of keys is: $n_{max-key} = t^h - 1$
- The space required to store the tree is: $O(n)$
- Inserting a record requires: $O(Log_t\, n)$
- Performing a **point query**: $O(Log_t\, n)$
- Performing a **range query** with k elements: $O(Log_t\, n + k)$

نتیجه: جستجو و بازیابی رکورد مورد نظر یعنی **پرس و جوی نقطه‌ای(Point query یا Equality query)** در ساختار B^+ Tree از مرتبه $O(Log_t\, n)$ است. و پیدا کردن رکورد بعدی در ساختار B^+ Tree از مرتبه $o(1)$ است. همچنین جستجو و بازیابی رکوردهای مورد نظر بازه‌ای یعنی **پرس و جوی بازه‌ای(Range query)** در ساختار B^+ Tree از مرتبه $O(Log_t\, n + k)$ است. پرس و جوی بازه‌ای جلوتر شرح داد می‌شود.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

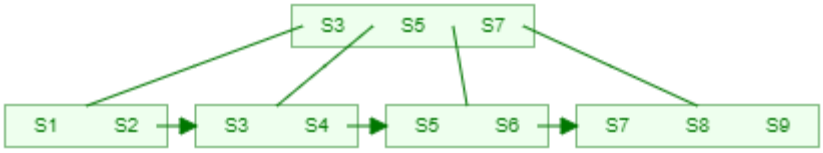
S1,S2,S3,S4,S5,S6,S7,S8,S9

توجه: فرض کنید درجه t=4 و به تبع حداکثر تعداد کلید جستجو در هر گره t-1 یعنی 4-1=3 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=3	
Number of Pointers	t=4	

ساختار نهایی B^+ Tree **به صورت زیر است:**



مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

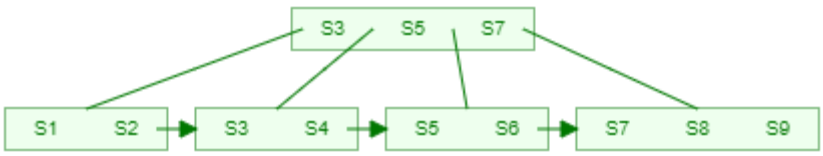
S1,S2,S3,S4,S5,S6,S7,S8,S9

توجه: فرض کنید درجه t=5 و به تبع حداکثر تعداد کلید جستجو در هر گره t-1 یعنی 5-1=4 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=4	
Number of Pointers	t=5	

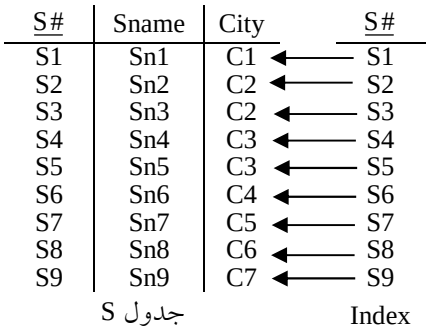
ساختار نهایی B^+ Tree **به صورت زیر است:**



نتیجه: دو رشته‌ی یکسان از نظر مقادیر در دو مثال قبل؛ و یکسان از نظر ترتیب درج و متفاوت از نظر درجه، دو شکل یکسان در ساختار، چیدمان و تعداد سطوح داشتند.

پیاده‌سازی شاخص فوق با ساختار آرایه

در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه به دلیل مزایای ساختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.



توجه: رکورد S9 پس از جستجو در ساختار آرایه و به صورت خطی در 9 حرکت در رکورد نهم یافت می‌شود، که مرتبه اجرایی آن $O(n)$ است.

توجه: از آن‌جا که **طول رکورد** در ساختار شاخص نسبت به جدول پایه کوچکتر است، واضح است که سرعت جستجو و بازیابی روی ساختار شاخص نسبت به جدول پایه بیش‌تر است. به بیان دیگر زمان اجرای جستجو و بازیابی کاهش می‌یابد. مطابق رابطه‌ی زیر:

$$\text{زمان انتقال رکورد } (T_f) = \frac{\text{طول رکورد } (L)}{\text{نرخ انتقال } (R)}$$

توجه: رابطه‌ی فوق در شاخص‌های با ساختار آرایه و ساختار B^+ Tree صادق است.

۲-پرس و جوی بازه‌ای(Range query)

در پرس و جوی بازه‌ای، هدف شناسایی و بازیابی سطرهایی است که مقدار ستون مورد جستجو در **یک بازه** مشخص قرار دارد.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

S1,S2,S3,S4,S5,S6,S7,S8,S9

پرس و جوی زیر را در نظر بگیرید:

```
SELECT S#
FROM S
WHERE S#>'S4' AND S#<='s9'
```

شرط جلوی where یعنی S#>'S4' AND S#<='s9' **یک بازه** مشخص است و این یعنی **پرس و جوی بازه‌ای**. خروجی پرس و جوی فوق به صورت زیر است:

S#
S5
S6
S7
S8
S9

توجه: رکوردهای S5، S6، S7، S8 و S9 پس از جستجو در ساختار B^+ Tree و به صورت درختی در 4 حرکت در سطح چهارم یافت می‌شود، که مرتبه اجرایی آن $O(Log_t n+k)$ است.

پیاده‌سازی شاخص فوق با ساختار B^+ Tree

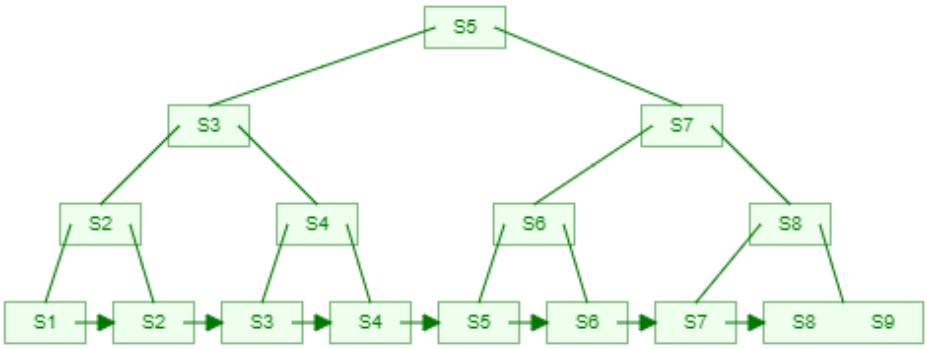
در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه بـه دلیـل مزایـای سـاختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.

توجه: فرض کنید درجه t=3 و به تبع حداکثر تعداد کلیـد جسـتجو در هـر گـره t-1 یعنـی 3-1=2 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=2	
Number of Pointers	t=3	

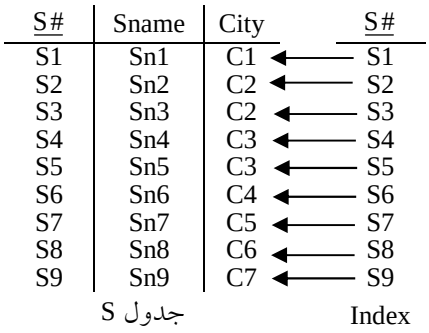
ساختار نهایی B^+ Tree به صورت زیر است:



نتیجه: رکوردهای S5، S6، S7، S8 و S9 پس از جستجو در ساختار B^+ Tree و به صورت درختی در 4 حرکت و چند قدم در سطح چهارم یافت می‌شود، که مرتبه اجرایی آن $O(Log_t n+k)$ است.

پیاده‌سازی شاخص فوق با ساختار آرایه

در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه بـه دلیـل مزایـای سـاختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.



توجه: رکوردهای S5، S6، S7، S8 و S9 پس از جستجو در ساختار آرایه و به صورت خطی در 9 حرکت در رکورد نهم یافت می‌شود، که مرتبه اجرایی آن $O(n)$ است.

شاخص مرتب‌شده به صورت Nonclustered Index

توجه: هر جدول می‌تواند حداکثر نهصد و نود و نه Nonclustered Index داشته باشد.

توجه: در Nonclustered Index ترتیب منطقی رکوردها در جدول پایه با ترتیب فیزیکی چیدمان آن‌ها بر روی هارد دیسک لزوما یکسان نیست والزامی هم برای آن وجود ندارد. و تنها یک مقدار و اشاره‌گر به رکوردی که حاوی مقدار مورد نظر در جدول پایه است در Nonclustered Index نگهداری می‌شود.

توجه: به شاخص مرتب‌شده به صورت Nonclustered Index، شاخص فرعی یا ثانویه نیز می‌گویند.

توجه: مقادیر شاخص مرتب‌شده به صورت Nonclustered Index می‌تواند یکتا و منحصر به فرد باشد اگر UNIQUE تعریف گردد و می‌تواند یکتا و منحصر به فرد نباشد اگر UNIQUE تعریف نگردد.

قطعه کد ایجاد Nonclustered Index به صورت زیر است:

```
CREATE [UNIQUE] [NONCLUSTERED] INDEX index_name
ON table_name (column1 [ASC | DESC], column2 [ASC | DESC], ...);
```

توجه: به طور پیش فرض و قابل تغییر در ایجاد Nonclustered Index دستور UNIQUE استفاده نمی‌گردد، که می‌شود **Non-Unique, Non-Clustered** در صورت نیاز به یکتایی و منحصر به فرد بودن مقادیر شاخص دستور UNIQUE استفاده می‌گردد، که می‌شود **Unique, Non-Clustered**.

توجه: به طور پیش فرض و قابل تغییر در ایجاد Nonclustered Index دستور Ascending یا ASC استفاده می‌گردد، تا **صعودی** بودن مقادیر شاخص **گارانتی** گردد. در صورت نیاز به **نزولی** بودن مقادیر شاخص دستور Descending یا DESC استفاده می‌گردد.

توجه: اگر پشت دستور INDEX نه دستور clustered و نه دستور Nonclustered باشد، حالت

پیش فرض Nonclustered است.

توجه: اگر پشت دستور INDEX دستور Unique نباشد، حالت پیش فرض Non-Unique است که نوشته هم نمی‌شود.

مثال: با اجرای قطعه کد زیر جدول S با کلید اصلی S# و به تبع یک clustered Index روی ستون S# ایجاد می‌گردد.

```
CREATE TABLE S (
    S# int NOT NULL,
    SName varchar(255),
    City varchar(255),
    PRIMARY KEY (S#)
);
```

قطعه کد ایجاد clustered Index که بطور خودکار اجرا می‌شود، به صورت زیر است:

```
CREATE UNIQUE CLUSTERED INDEX PK_S
ON S (S# ASC);
```

جداول زیر را در نظر بگیرید:

S#	Sname	City
S1	Sn1	C1
S2	Sn2	C2
S3	Sn3	C2
S4	Sn4	C3
S5	Sn5	C4
S6	Sn6	C5
S7	Sn7	C6
S8	Sn8	C7
S9	Sn9	C7

جدول S

S#	P#	QTY
S1	P1	10
S1	P2	20
S2	P1	30

جدول SP

P#	Pname	Color
P1	Pn1	Red
P2	Pn2	Blue

جدول P

با استفاده از clustered Index پیش فرض و موجود روی ستون S#، سرعت پاسخگویی به تمامی پرس و جوهایی که جستجو را بر اساس S# در جدول S انجام می‌دهند، افزایش می‌یابد.

مثال:

```
SELECT S#
FROM S
WHERE S#='S9'
```

اما در مورد پرس و جوهایی که جستجو را بر اساس ستون S# انجام نمی‌دهند. تعریف شاخص فوق هیچ تاثیری در سرعت پاسخگویی به پرس و جو ندارد.

مثال: اجرای دستور زیر موجب ایجاد یک Nonclustered Index و Non-Unique به صورت **صعودی**، به نام CityX روی ستون City در جدول S می‌شود.

```
CREATE NONCLUSTERED INDEX CityX
ON S (City ASC);
```

با توجه به دستورات پیش فرض، قطعه کد زیر، معادل قطعه کد فوق است:

```
CREATE INDEX CityX
ON S (City);
```

با استفاده از Nonclustered Index، سرعت پاسخگویی به تمامی پرس و جوهایی که جستجو را بر اساس City در جدول S انجام می‌دهند، افزایش می‌یابد.

مثال:

```
SELECT S#
FROM S
WHERE City='C2'
```

اما در مورد پرس و جوهایی که جستجو را بر اساس ستون City انجام نمی‌دهند. تعریف شاخص فوق هیچ تاثیری در سرعت پاسخگویی به پرس و جو ندارد.

مثال: اجرای دستور زیر موجب ایجاد یک Nonclustered Index و Unique به صورت **صعودی**، به نام SnameX روی ستون Sname در جدول S می‌شود.

```
CREATE UNIQUE NONCLUSTERED INDEX SnameX
ON S (Sname ASC);
```

با توجه به دستورات پیش فرض، قطعه کد زیر، معادل قطعه کد فوق است:

```
CREATE UNIQUE INDEX SnameX
ON S (Sname);
```

توجه: دستور فوق یعنی ایجاد شاخص یکتا توسط دستور Unique فقط در صورتی قابل اجرا توسط DBMS روی ستون مورد نظر است، که از قبل ستون مورد نظر مقادیر تکراری نداشته باشد، در غیر اینصورت یعنی وجود مقادیر تکراری در ستون مورد نظر، دستور فوق از سوی DBMS رد می‌شود. که این می‌شود پیشا اجرای دستور فوق، همچنین پس از ایجاد دستور فوق یعنی ایجاد شاخص یکتا بر روی ستون حائز شرایط ایجاد شاخص یکتا، یکتا بودن ستون مورد نظر، تا مادامی که شاخص یکتا وجود دارد، **گارانتی** می‌شود که این می‌شود پسااجرای دستور فوق.

با استفاده از Nonclustered Index، سرعت پاسخگویی به تمامی پرس و جوهایی که جستجو را بر اساس Sname در جدول S انجام می‌دهند، افزایش می‌یابد.

مثال:

```
SELECT S#
FROM S
WHERE Sname='Sn9'
```

اما در مورد پرس و جوهایی که جستجو را بر اساس ستون Sname انجام نمی‌دهند. تعریف شاخص فوق هیچ تاثیری در سرعت پاسخگویی به پرس و جو ندارد.

به طور کلی دو نوع پرس و جو در جداول پایگاه داده انجام می‌گردد:

(۱)پرس و جوی نقطه‌ای و (۲) پرس و جوی بازه‌ای که در ادامه به بررسی آن می‌پردازیم.

۱-پرس و جوی نقطه‌ای(Point query یا Equality query)

در پرس و جوی نقطه‌ای، هدف شناسایی و بازیابی سطرهایی است که مقدار ستون مورد جستجو در **یک مقدار** مشخص قرار دارد.

اجرای دستور زیر موجب ایجاد یک Nonclustered Index و Non-Unique به صورت **صعودی**، به نام CityX روی ستون City در جدول S می‌شود.

```
CREATE NONCLUSTERED INDEX CityX
ON S (City ASC);
```

با توجه به دستورات پیش فرض، قطعه کد زیر، معادل قطعه کد فوق است:

```
CREATE INDEX CityX
ON S (City);
```

جداول زیر را در نظر بگیرید:

S#	Sname	City
S1	Sn1	C1
S2	Sn2	C2
S3	Sn3	C2
S4	Sn4	C3
S5	Sn5	C4
S6	Sn6	C5
S7	Sn7	C6
S8	Sn8	C7
S9	Sn9	C7

جدول S

S#	P#	QTY
S1	P1	10
S1	P2	20
S2	P1	30

جدول SP

P#	Pname	Color
P1	Pn1	Red
P2	Pn2	Blue

جدول P

توجه: با استفاده از Nonclustered Index، سرعت پاسخگویی به تمامی پرس و جوهایی که جستجو را بر اساس City در جدول S انجام می‌دهند، افزایش می‌یابد.

توجه: اما در مورد پرس و جوهایی که جستجو را بر اساس ستون City انجام نمی‌دهند. تعریف شاخص فوق هیچ تاثیری در سرعت پاسخگویی به پرس و جو ندارد.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:(ascending یعنی صعودی)

C1,C2,C2,C3,C4,C5,C6,C7,C7

پرس و جوی زیر را در نظر بگیرید:

```
SELECT S#
FROM S
WHERE City='C7'
```

توجه: شرط جلوی where یعنی City='C7' **یک مقدار** مشخص است و این یعنی **پرس و جوی نقطه‌ای**. خروجی پرس و جوی فوق به صورت زیر است:

S#
S8
S9

پیاده‌سازی شاخص فوق با ساختار B^+ Tree

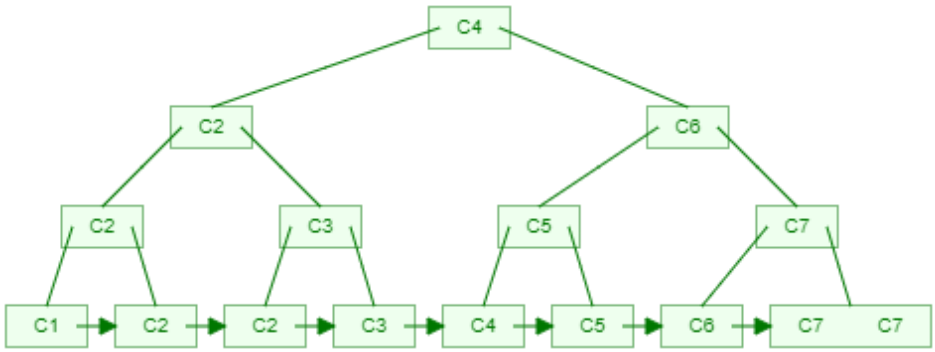
در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه بـه دلیـل مزایـای سـاختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.

توجه: فرض کنید درجه t=3 و به تبع حداکثر تعداد کلیـد جسـتجو در هـر گـره t-1 یعنـی 3-1=2 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=2	
Number of Pointers	t=3	

ساختار نهایی B^+ Tree به صورت زیر است:



نتیجه: رکورد C7 پس از جستجو در ساختار B^+ Tree و به صورت درختی در 4 حرکت در سطح چهارم یافت می‌شود، که مرتبه اجرایی آن $O(\text{Log}_t\, n)$ است.

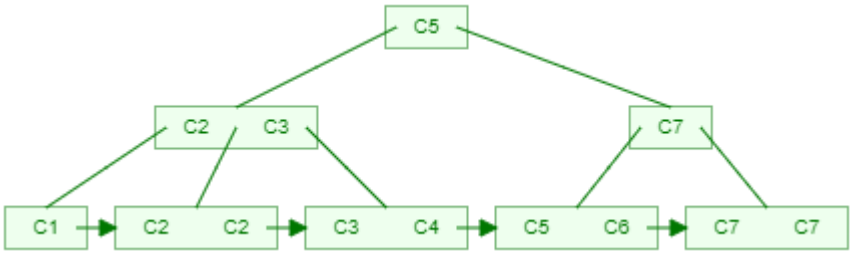
مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:(descending یعنی نزولی)
C7,C7,C6,C5,C4,C3,C2,C2,C1

توجه: فرض کنید درجه t=3 و به تبع حداکثر تعداد کلیـد جسـتجو در هـر گـره t-1 یعنـی 3-1=2 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=2	
Number of Pointers	t=3	

ساختار نهایی B^+ Tree **به صورت زیر است:**



نتیجه: دو رشته‌ی یکسان از نظر مقادیر؛ و متفاوت از نظر ترتیب درج و یکسان از نظر درجه، دو شکل متفاوت در ساختار، چیدمان و تعداد سطوح داشتند.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

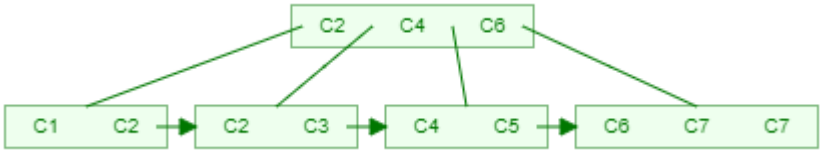
C1,C2,C2,C3,C4,C5,C6,C7,C7

توجه: فرض کنید درجه t=4 و به تبع حداکثر تعداد کلید جستجو در هر گره t-1 یعنی 4-1=3 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=3	
Number of Pointers	t=4	

ساختار نهایی B^+ Tree **به صورت زیر است:**



مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

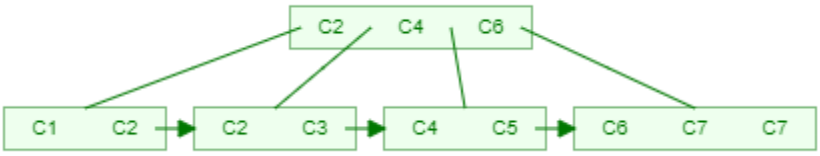
C1,C2,C2,C3,C4,C5,C6,C7,C7

توجه: فرض کنید درجه t=5 و به تبع حداکثر تعداد کلید جستجو در هر گره t-1 یعنی 5-1=4 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=4	
Number of Pointers	t=5	

ساختار نهایی B^+ Tree به صورت زیر است:



نتیجه: دو رشته‌ی یکسان از نظر مقادیر در دو مثال قبل؛ و یکسان از نظر ترتیب درج و متفاوت از نظر درجه، دو شکل یکسان در ساختار، چیدمان و تعداد سطوح داشتند.

پیاده‌سازی شاخص فوق با ساختار آرایه

در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه به دلیل مزایای ساختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.

S#	Sname	City
S1	Sn1	C1
S2	Sn2	C2
S3	Sn3	C2
S4	Sn4	C3
S5	Sn5	C4
S6	Sn6	C5
S7	Sn7	C6
S8	Sn8	C7
S9	Sn9	C7

جدول S

City
C1
C2
C2
C3
C4
C5
C6
C7
C7

Index

توجه: رکورد C7 پس از جستجو در ساختار آرایه و به صورت خطی در 9 حرکت در رکورد هشتم و نهم یافت می‌شود، که مرتبه اجرایی آن $O(n)$ است.

اجرای دستور زیر موجب ایجاد یک Nonclustered Index و Unique به صورت **صعودی**، به نـام SnameX روی ستون Sname در جدول S می‌شود.

```
CREATE UNIQUE NONCLUSTERED INDEX SnameX
ON S (Sname ASC);
```

با توجه به دستورات پیش فرض، قطعه کد زیر، معادل قطعه کد فوق است:

```
CREATE UNIQUE INDEX SnameX
ON S (Sname);
```

جداول زیر را در نظر بگیرید:

S#	Sname	City
S1	Sn1	C1
S2	Sn2	C2
S3	Sn3	C2
S4	Sn4	C3
S5	Sn5	C4
S6	Sn6	C5
S7	Sn7	C6
S8	Sn8	C7
S9	Sn9	C7

جدول S

S#	P#	QTY
S1	P1	10
S1	P2	20
S2	P1	30

جدول SP

P#	Pname	Color
P1	Pn1	Red
P2	Pn2	Blue

جدول P

توجه: دستور فوق یعنی ایجاد شاخص یکتا توسط دستور Unique فقـط در صـورتی قابـل اجـرا توسط DBMS روی ستون مورد نظر است، که از قبل ستون مورد نظر مقادیر تکراری نداشته باشد، در غیر اینصورت یعنی وجود مقادیر تکراری در ستون مورد نظر، دستور فوق از سـوی DBMS رد می‌شود. که این می‌شود پیشا اجرای دستور فوق، همچنین پس از ایجاد دسـتور فـوق یعنـی ایجـاد شاخص یکتا بر روی ستون حائز شرایط ایجاد شاخص یکتا، یکتا بودن ستون مورد نظر، تا مـادامی که شاخص یکتا وجود دارد، گارانتی می‌شود که این می‌شود پسااجرای دستور فوق.

توجه: با استفاده از Nonclustered Index، سرعت پاسـخگویی بـه تمـامی پـرس و جوهـایی کـه جستجو را بر اساس Sname در جدول S انجام می‌دهند، افزایش می‌یابد.

توجه: اما در مورد پرس و جوهایی که جستجو را بر اساس ستون Sname انجام نمی‌دهند. تعریف شاخص فوق هیچ تاثیری در سرعت پاسخگویی به پرس و جو ندارد.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:(ascending یعنی صعودی)

Sn1,Sn2,Sn3,Sn4,Sn5,Sn6,Sn7,Sn8,Sn9

پرس و جوی زیر را در نظر بگیرید:

```
SELECT S#
FROM S
WHERE Sname='Sn9'
```

توجه: شرط جلوی where یعنی Sname='Sn9' **یک مقدار** مشخص است و این یعنی **پرس و جوی نقطه‌ای**. خروجی پرس و جوی فوق به صورت زیر است:

S#
S9

پیاده‌سازی شاخص فوق با ساختار B^+ Tree

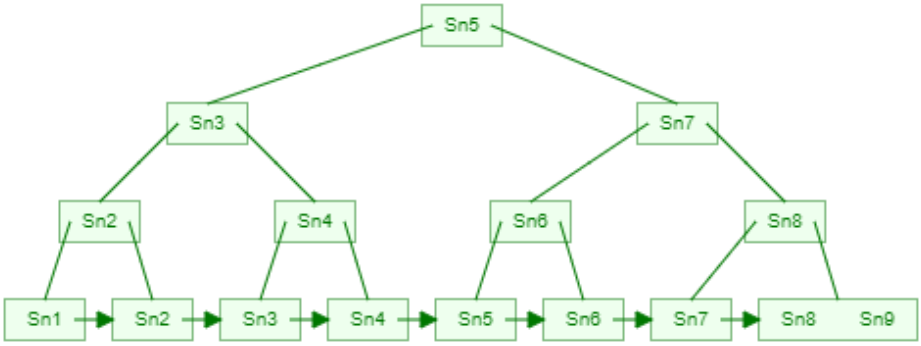
در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه بـه دلیـل مزایـای سـاختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.

توجه: فرض کنید درجه t=3 و به تبع حداکثر تعداد کلیـد جسـتجو در هـر گـره t-1 یعنـی 3-1=2 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=2	
Number of Pointers	t=3	

ساختار نهایی B^+ Tree به صورت زیر است:



نتیجه: رکورد Sn9 پس از جستجو در ساختار B^+ Tree و به صورت درختی در 4 حرکت در سطح چهارم یافت می‌شود، که مرتبه اجرایی آن $O(\text{Log}_t\, n)$ است.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:(descending یعنی نزولی)

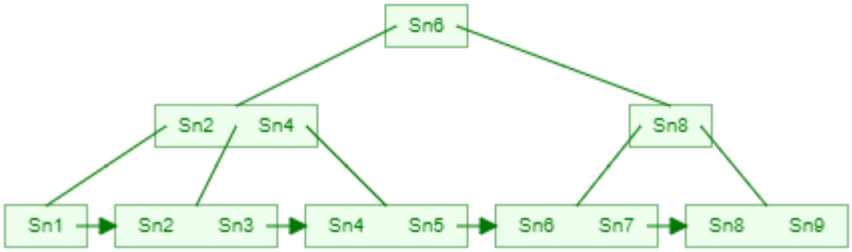
Sn9,Sn8,Sn7,Sn6,Sn5,Sn4,Sn3,Sn2,Sn1

توجه: فرض کنید درجه t=3 و به تبع حداکثر تعداد کلیـد جسـتجو در هـر گـره t-1 یعنـی 3-1=2 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=2	
Number of Pointers	t=3	

ساختار نهایی B^+ Tree **به صورت زیر است:**



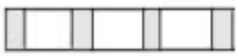
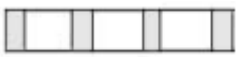
نتیجه: دو رشته‌ی یکسان از نظر مقادیر؛ و متفاوت از نظر ترتیب درج و یکسان از نظـر درجـه، دو شکل متفاوت در ساختار، چیدمان و تعداد سطوح داشتند.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

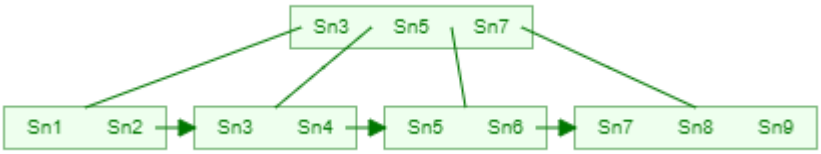
Sn1,Sn2,Sn3,Sn4,Sn5,Sn6,Sn7,Sn8,Sn9

توجه: فرض کنید درجه t=4 و به تبع حداکثر تعداد کلیـد جسـتجو در هـر گـره t-1 یعنـی 4-1=3 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=3	
Number of Pointers	t=4	

ساختار نهایی B^+ Tree **به صورت زیر است:**



مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

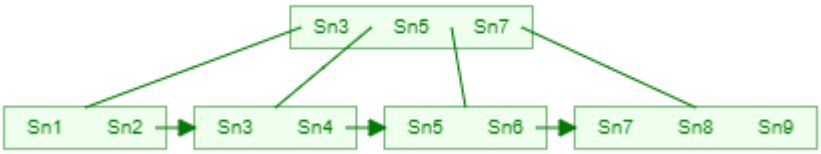
Sn1,Sn2,Sn3,Sn4,Sn5,Sn6,Sn7,Sn8,Sn9

توجه: فرض کنید درجه t=5 و به تبع حداکثر تعداد کلید جستجو در هر گره t-1 یعنی 5-1=4 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=4	
Number of Pointers	t=5	

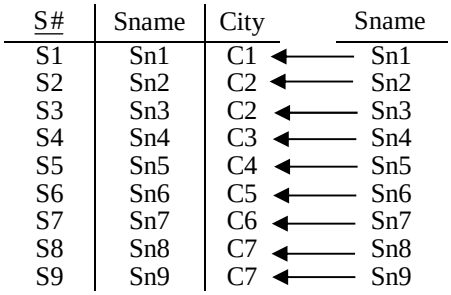
ساختار نهایی B^+ Tree به صورت زیر است:



نتیجه: دو رشته‌ی یکسان از نظر مقادیر در دو مثال قبل؛ و یکسان از نظر ترتیب درج و متفاوت از نظر درجه، دو شکل یکسان در ساختار، چیدمان و تعداد سطوح داشتند.

پیاده‌سازی شاخص فوق با ساختار آرایه

در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه به دلیل مزایای ساختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.



توجه: رکورد Sn9 پس از جستجو در ساختار آرایه و به صورت خطی در 9 حرکت در رکورد نهم یافت می‌شود، که مرتبه اجرایی آن $O(n)$ است.

۲-پرس و جوی بازه‌ای(Range query)

در پرس و جوی بازه‌ای، هدف شناسایی و بازیابی سطرهایی است که مقدار ستون مورد جستجو در **یک بازه** مشخص قرار دارد.

مثال: درج رشته زیر را بر اساس جدول S در نظر بگیرید:

Sn1,Sn2,Sn3,Sn4,Sn5,Sn6,Sn7,Sn8,Sn9

پرس و جوی زیر را در نظر بگیرید:

```
SELECT S#
FROM S
WHERE Sname>'Sn4' AND Sname<='Sn9'
```

شرط جلوی where یعنی Sname>'Sn4' AND Sname<='Sn9' **یک بازه** مشخص است و این یعنی **پرس و جوی بازه‌ای**. خروجی پرس و جوی فوق به صورت زیر است:

S#
Sn5
Sn6
Sn7
Sn8
Sn9

توجه: رکوردهای Sn5، Sn6، Sn7، Sn8 و Sn9 پس از جستجو در ساختار B^+ Tree و به صورت درختی در 4 حرکت در سطح چهارم یافت می‌شود، که مرتبه اجرایی آن $O(Log_t n+k)$ است.

پیاده‌سازی شاخص فوق با ساختار B^+ Tree

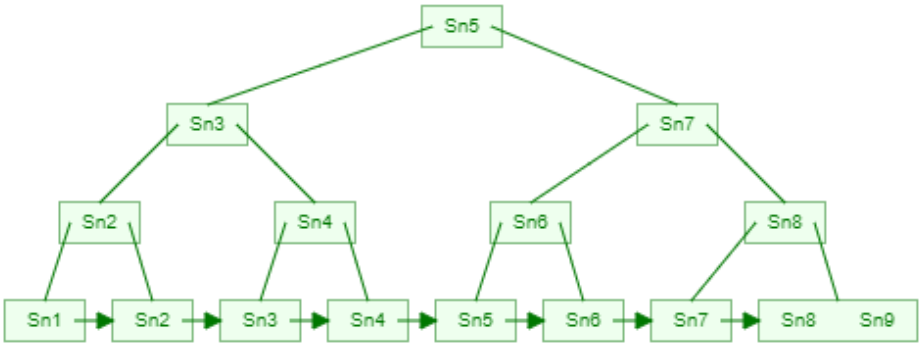
در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه بـه دلیـل مزایـای سـاختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.

توجه: فرض کنید درجه t=3 و به تبع حداکثر تعداد کلید جستجو در هر گره t-1 یعنی 3-1=2 است.

ساختار حداکثر درجه(اشاره‌گرها) و حداکثر کلید جستجو به صورت زیر است:

Number of Keys	key=2	
Number of Pointers	t=3	

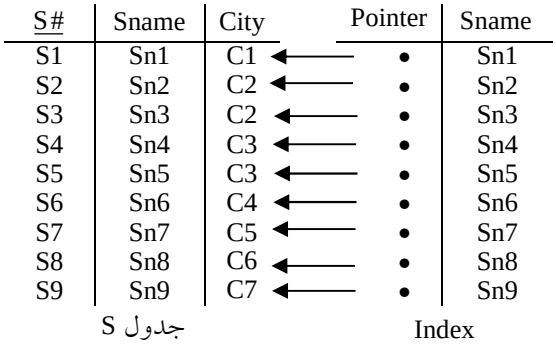
ساختار نهایی B^+ Tree به صورت زیر است:



نتیجه: رکوردهای Sn5، Sn6، Sn7، Sn8 وSn9 پس از جستجو در ساختار B^+ Tree و به صورت درختی در 4 حرکت و چند قدم در سطح چهارم یافت می‌شود، که مرتبه اجرایی آن $O(Log_t\, n+k)$ است.

پیاده‌سازی شاخص فوق با ساختار آرایه

در گذشته ساختار شاخص، به کمک آرایه پیاده‌سازی می‌شد. اما امروزه به دلیل مزایای ساختار درختی و به تبع سرعت بیشتر در جستجو و بازیابی اطلاعات، ساختار B^+ Tree جهت پیاده‌سازی شاخص مورد استفاده قرار می‌گیرد.



توجه: رکوردهای Sn5، Sn6، Sn7، Sn8 و Sn9 پس از جستجو در ساختار آرایه و به صورت خطی در 9 حرکت در رکورد نهم یافت می‌شود، که مرتبه اجرایی آن $O(n)$ است.

۲- شاخص از نوع Hash(Hash Index)

شاخص از نوع Hash به کمک ساختار فایل مستقیم(تکنیک درهم‌سازی) قابل پیاده‌سازی است.

توجه: در SQL Server شاخص از نوع Hash پیاده‌سازی نشده است.

درهم‌سازی(Hashing)

منظور از درهم‌سازی، تبدیل کلید به آدرس (Key to Address Transformation=KAT) توسط پردازشی است که بر روی کلید انجام می‌گیرد. به عبارتی دیگر تابع درهم‌ساز (hashing function) تابعی است که هنگام **درج** یک رکورد، کلید جستجو که یکی از صفات است را گرفته، پردازش و یا محاسباتی را بر روی آن انجام داده و آدرس معادل آنرا بر می‌گرداند. این آدرس همان جایی است که رکورد مورد نظر باید در آن قرار گیرد و درج شود و به آن آدرس طبیعی(Natural Address)، حفره طبیعی(Natural Slot) و یا آدرس خانگی (Home Address) نیز می‌گویند. هنگام جستجو و بازیابی رکورد مورد نظر همین عمل مجددا صورت می‌گیرد، یعنی مجددا کلید مربوطه به تابع درهم‌ساز داده می‌شود و سیستم آدرس ذخیره‌سازی رکورد مرتبط با آنرا همانند قبل تولید می‌کند و با این تکنیک، دستیابی مستقیم به رکوردها امکان‌پذیر می‌گردد. به تابع درهم‌ساز، تابع مبدل یا نگاشتگر (Mapping Function) نیز گفته می‌شود.

توجه: همانطور که پیش‌تر گفتیم، در ساختار آرایه جستجو و بازیابی نقطه‌ای و بازه‌ای از مرتبه $O(n)$ بود. همچنین در ساختار B^+ Tree جستجو و بازیابی نقطه‌ای(Equality Query) از مرتبه $O(Log_t n)$ و جستجو و بازیابی بازه‌ای(Range Query) از مرتبه $O(Log_t n+k)$ بود. اما در ساختار فایل مستقیم(تکنیک درهم‌سازی) جستجو و بازیابی نقطه‌ای(Equality Query) از مرتبه $O(1)$ است. دستیابی از مرتبه $O(1)$ به جدول پایه، به این معناست که بدون توجه، به اندازه جدول پایه، جهت دسترسی به یک رکورد دلخواه، همواره تعداد اندکی پیگرد نیاز است. در حالت ایده‌آل

با یک حرکت می‌توان به رکورد مورد نظر دسترسی پیدا کرد. همچنین در ساختار فایل مستقیم(تکنیک درهم‌سازی) جستجو و بازیابی بازه‌ای(Range Query) می‌بایست برای تک تک اعضای بازه به طور مستقل تابع hash اجرا شود که این امر مستلزم صرف وقت و هزینه زیادی است. فایل مستقیم بی‌نظم است و امکان پردازش سریالی و ترتیبی را ندارد، زیرا فایل مستقیم بر حسب کلید مرتب نشده است. به عبارت دیگر به علت درهم بودن فایل، واکشی رکورد بعدی مشابه واکشی یک رکورد جدید است، پس امکان پردازش سریالی و ترتیبی در آن وجود ندارد، ساختار فایل مستقیم مناسب محیط‌هایی است که دستیابی سریع به رکوردها مورد نیاز است و پردازش‌ها ترتیبی و سریالی مد نظر نباشد. بنابراین برای جستجو و بازیابی بازه‌ای(Range Query)، شاخص‌گذاری مرتب شده با ساختار B^+ Tree با مرتبه $O(Log_t n+k)$ مناسب‌تر است. و شاخص‌های از نوع Hash برای پاسخ به Range Query ها مفید نیست.

توجه: در ساختار شاخص از نوع Hash، شاخص دارای یک فضای آدرس(Address Space) است با m آدرس از آدرس 1 تا m یا از صفر تا m-1 و هر آدرس مربوط است به یک حفره و هر حفره، مکان ذخیره‌سازی یک رکورد است. اگر تعداد رکوردها n باشد، آنگاه m>=n است. توابع درهم‌ساز(hashing) رکوردها را به صورت تصادفی و نامنظم در فضای آدرس، پخش می‌کنند.

پدیده تصادم(برخورد Collision)

اگر دو کلید متمایز، پس از اعمال تابع مبدل، آدرس‌های مساوی تولید کنند، یعنی $k_i \neq k_j \Rightarrow a_i = a_j$، آنگاه تصادم رخ داده است. که پس از وقوع پدیده برخود می‌بایست مساله برخورد به نحوی حل شود.

مثال: نگهداری آدرس n=6 رکورد در شاخص، کلید رکوردها **بخش عددی** ستون #S از جدول پایه S و تعداد سطرهای آدرس دهی در شاخص برابر m=14 است. تابع مولد یعنی h به صورت زیر است:

$h(S\#) \Rightarrow address = S\# \bmod 13$

$R_1 : h(S100) \Rightarrow address = 100 \bmod 13 = 9$

$R_2 : h(S200) \Rightarrow address = 200 \bmod 13 = 5$

$R_3 : h(S300) \Rightarrow address = 300 \bmod 13 = 1$

$R_4 : h(S400) \Rightarrow address = 400 \bmod 13 = 10$

$R_5 : h(S500) \Rightarrow address = 500 \bmod 13 = 6$

$R_6 : h(S1400) \Rightarrow address = 1400 \bmod 13 = 9$

next	Pointer	S#	Sname	City	Pointer	m	Keys
					×	0	
		S300	Sn3	C3	•	1	300
					×	2	
					×	3	
					×	4	
		S200	Sn2	C2	•	5	200
		S500	Sn5	C5	•	6	500
					×	7	
					×	8	
	•	S100	Sn1	C1	•	9	100,1400
		S400	Sn4	C4	•	10	400
					×	11	
					×	12	
					×	13	

Index

توجه: رکورد R_6 هنگام درج با رکورد R_1 تصادم دارد، که پس از وقوع پدیده برخود می‌بایست مساله برخورد به نحوی حل شود. برای مثال رکورد R_1 توسط اشاره‌گر به رکورد R_6 اشاره کند که این موضوع می‌تواند برای یک حفره مدام تکرار شود و یک زنجیره تشکیل شود. و یا رکورد R_6 در اولین حفره خالی مثل حفره 11 درج شود.

توجه: راه حل ایده‌آل برای حل مساله برخورد، آن است که از الگوریتم و تابعی استفاده شود که به طور کلی از تصادم جلوگیری کند. به چنین تابعی، **تابع درهم‌ساز کامل** گفته می‌شود. در عمل پیدا کردن چنین تابعی بسیار پیچیده و زمان‌بر است. یعنی با آنکه پیدا کردن تابع درهم‌سازی کامل امکان‌پذیر است ولی هزینه‌ی یافتن آن به قدری زیاد است که در عمل هیچگاه از این روش استفاده نمی‌گردد.

راه حل عمومی آن است که:

۱-ابتدا با روش‌هایی سعی کنیم تعداد برخوردها به میزان قابل توجهی کاهش پیدا کند.

۲-سپس راه حل‌های مناسبی را بیابیم که اگر برخوردی رخ داد، آنرا بر طرف سازد.

توجه: تابع مولد(درهم‌ساز) همواره باید، به ازای یک کلید معین، همان آدرسی را تولید کند که در بار اولیه ساخته است. به عبارت دیگر تابع درهم‌ساز باید توانایی **تکرارشدنی** داشته باشد. همچنین تابع مولد ایده‌آل باید اول اینکه رکوردها را به طور یکنواخت توزیع کند، دوم اینکه باید از تمام اجزای کلید استفاده کند و سوم اینکه تعداد برخوردها را نیز کاهش دهد.

خصوصیات توابع مبدل ایده‌آل

۱-پراکنده کردن رکوردها: توابع مبدل(درهم‌ساز) باید رکوردها رو به صورت اتفاقی و تصادفی بین آدرس‌ها توزیع کند. بدیهی است که اگر احتمال قرارگیری کلیدها در خانه‌های مشخصی بیشتر

از سایر خانه‌ها باشد، در آنجا پدیده تصادم بیشتر رخ می‌دهد. ولی هرچقدر که کلیدها در فضای آدرس، بیشتر پخش شوند آنگاه احتمال برخورد نیز کمتر می‌شود. برای مثال اگر تابع مولد جهت درهم‌سازی فقط حرف اول کلید جستجو را معیار درهم‌سازی قرار دهد، آنگاه از آنجا که برای مثال نام‌های متعددی با حرف A یا M آغاز شده و اسامی کمی با حرف Z و G شروع می‌شوند، پس این تابع مولد به طور مناسب رکوردها را پراکنده نمی‌سازد. بنابراین همانطور که گفتیم، تابع مولد ایده‌آل باید رکوردها را به طور یکنواخت توزیع کند، باید از تمام اجزای کلید استفاده کند و تعداد برخوردها را نیز کاهش دهد.

۲-رشد خطی فضای آدرس: بدیهی است که اگر تعداد کمی رکورد را بخواهیم در بین تعداد زیادی آدرس قرار دهیم، تابع درهم‌ساز ساده‌تر بوده و احتمال برخورد نیز کاهش می‌یابد. برای مثال اگر بخواهیم 25 رکورد را در یک فضای 1000 خانه‌ای ذخیره کنیم(n=25 و m=1000). در این حالت احتمال بروز تصادم کاهش می‌یابد ولی روشن است که در این روش، فضای زیادی به هدر خواهد رفت. در بعضی سیستم‌ها m به صورت پویا رشد می‌کند. به عبارت دیگر فضای آدرس به صورت خطی گسترش داده می‌شود. برای مثال یک روش آن است که به ازای هر بار تصادم یک حفره به فضای آدرس اضافه شود.(m=m+1).

۳-دخیره بیش از یک رکورد در یک آدرس(تکنیک باکت‌بندی): می‌توان فایل شاخص را به گونه‌ای پیاده‌سازی کرد که هر آدرس آن بتواند چند رکورد را در خود ذخیره کند. برای مثال اگر اندازه یک باکت برابر 512 بایت باشد و اندازه هر رکورد برابر 128 بایت باشد، آنگاه می‌توان در هر باکت 4 رکورد را جای داد. به چنین فضاهایی که می‌توانند چند رکورد را به این شیوه ذخیره کنند، باکت (bucket) گفته می‌شود. اگر فایل مستقیم باکت‌بندی شود، آنگاه به جای آدرس حفره، آدرس باکت خواهیم داشت. از صفر تا M-1.

مثال: نگهداری آدرس n=6 رکورد در شاخص، کلید رکوردها **بخش عددی** ستون S# از جدول پایه S و تعداد رکوردهای آدرس‌دهی در شاخص برابر m=14 است. همچنین اندازه یک باکت برابر 512 بایت و اندازه هر رکورد برابر 256 بایت است یعنی فاکتور باکت‌بندی برابر 2 رکورد در یک باکت است. تابع مولد یعنی h به صورت زیر است:

$$h(S\#) \Rightarrow \text{address} = S\# \bmod 13$$

$R_1 : h(S100) \Rightarrow address = 100 \bmod 13 = 9$

$R_2 : h(S200) \Rightarrow address = 200 \bmod 13 = 5$

$R_3 : h(S300) \Rightarrow address = 300 \bmod 13 = 1$

$R_4 : h(S400) \Rightarrow address = 400 \bmod 13 = 10$

$R_5 : h(S500) \Rightarrow address = 500 \bmod 13 = 6$

$R_6 : h(S1400) \Rightarrow address = 1400 \bmod 13 = 9$

next	Pointer	S#	Sname	City	Pointer	M	Keys
					× ×	0	
		S300	Sn3	C3	• ×	1	300
					× ×	2	
					× ×	3	
					× ×	4	
		S200	Sn2	C2	• ×	5	200
		S500	Sn5	C5	• ×	6	500
					× ×	7	
					× ×	8	
		S100 S1400	Sn1 S14	C1 C14	• •	9	100 1400
		S400	Sn4	C4	• ×	10	400
					× ×	11	
					× ×	12	
					× ×	13	

Index

توجه: رکورد R_6 هنگام درج بدون تصادم با رکورد R_1 در باکت شماره 9 قرار می‌گیرد.

توجه: در باکت‌بندی ممکن است فضای به هدر رفته زیاد باشد.

توجه: هنگام واکشی یک رکورد، تابع درهم‌ساز شماره حفره باکت را مشخص می‌کند. و کل آن باکت در حافظه واکشی شده و سپس رکوردهای موجود در آن جهت یافتن رکورد مورد نظر در حافظه جستجو می‌شود.

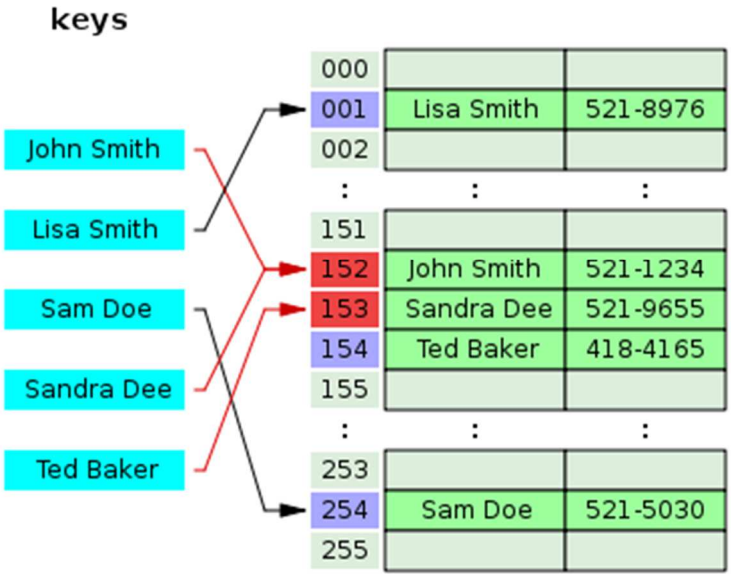
توجه: در حالت باکت‌بندی نیز همچنان مشکل سرریزی وجود دارد که برای مثال می‌توان از روش زنجیره اشاره‌گرها استفاده کرد. برای مثال حفره شماره 9 در مثال فوق دیگر جا ندارد ولی بدیهی است که تعداد سرریزها در حالت باکت‌بندی کمتر از حالت بدون باکت‌بندی است.

توجه: از نظر سیستم ورودی و خروجی، زمان انتقال یک رکورد با زمان انتقال یک باکت، تفاوت چندانی ندارد. بنابراین باکت‌بندی ایده مناسبی در جهت کاهش زمان انتقال و تصادم است.

روش‌های رفع مشکل برخورد

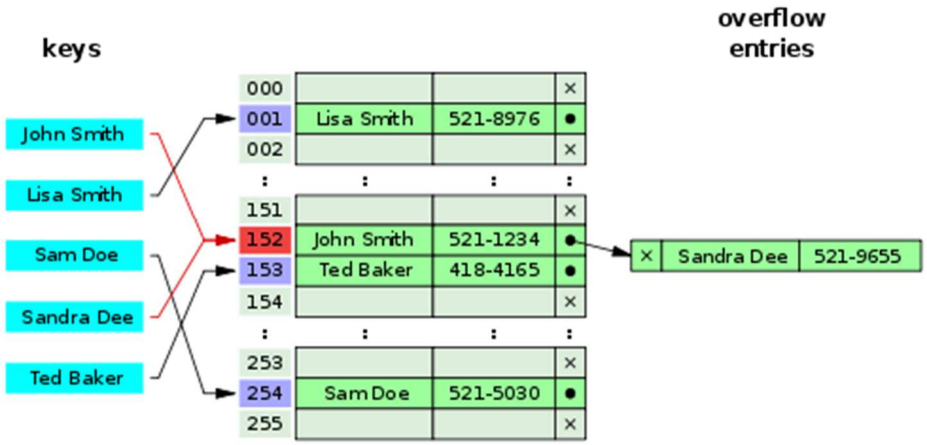
جستجوی خطی و درج در اولین حفره جادار(Linear Probing)

در این روش هنگام اضافه کردن رکورد در صورتی که برخورد رخ دهد، از نقطه‌ای که تصادم رخ داده است به سمت انتهای فایل به صورت خطی دنبال اولین حفره خالی جهت درج رکورد سرریز می‌گردیم. اگر تا انتهای فایل حفره‌ای پیدا نشد، بصورت چرخشی از ابتدای فایل تا نقطه تصادم جستجو را ادامه می‌دهیم. شکل زیر گویای مطلب است:

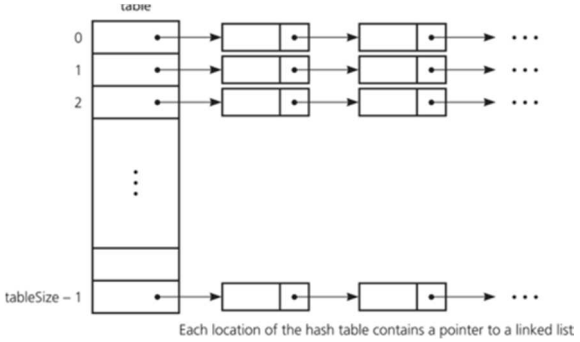


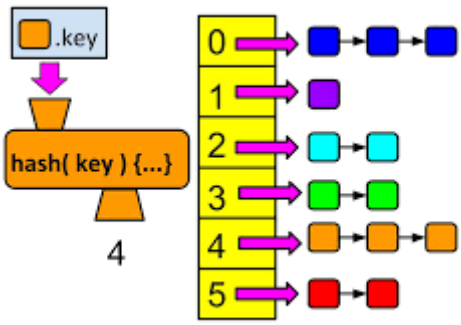
روش ساخت زنجیره اشاره‌گرها

در این روش هنگام اضافه کردن رکورد در صورتی که برخورد رخ دهد، اشاره‌گر جهت اتصال رکوردها به عنوان زنجیره اتصال مورد استفاده قرار می‌گیرد. شکل زیر گویای مطلب است:



دو تصویر زیر نیز گویای مطلب است:





تست‌های فصل هفتم: SQL دستورات DDL و DCL

۱- جدول T را در نظر بگیرید که روی ستون a، Clustered Index شده است. کدام مورد درست است؟ (مهندس کامپیوتر-دولتی ۹۷)

۱) اعمال سیاست شاخص‌گذاری، تاثیری بر حجم اطلاعات ذخیره‌شده بر روی دیسک ندارد.

۲) با اعمال سیاست شاخص‌گذاری، پاسخ به Range Queryهای مرتبط به a، با سرعت بیشتری انجام می‌شود.

۳) با اعمال سیاست شاخص‌گذاری، پاسخ به Equality Queryهای مرتبط به a، با سرعت کمتری انجام می‌شود.

۴) همه موارد درست هستند.

۲- در مورد شاخص‌گذاری (Indexing)، کدام مورد نادرست است؟ (مهندسی IT-دولتی ۹۷)

۱) B^+ Tree index برای پاسخ به Range Query مفید است.

۲) هر جدول می‌تواند حداکثر یک شاخص کلاستر شده داشته باشد.

۳) شاخص‌های از نوع Hash برای پاسخ به Range Query مفید هستند.

۴) هرچه تعداد شاخص‌های یک جدول بیشتر باشد، سرعت Insert در آن جدول کمتر می‌شود.

پاسخ تست‌های فصل هفتم: SQL دستورات DDL و DCL

۱- گزینه (۲) صحیح است.

دو هدف اصلی سیستم ذخیره و بازیابی اطلاعات در پایگاه داده‌ها، اول سرعت عملیات در ذخیره و بازیابی اطلاعات و دوم صرفه‌جویی در مصرف حافظه است. برای مثال کاهش افزونگی محتوایی (طبیعی) توسط نرمال‌سازی جداول منجر به کاهش میزان حافظه مصرفی می‌شود. عمل واکشی تک تک رکوردها وقت‌گیر است، برای رفع این عیب شاخص یا Index ابداع شد. برای اینکه بازیابی داده‌ها با سرعت و کارایی بیشتر صورت گیرد، از شاخص استفاده می‌شود. شاخص ساختمان داده‌ای است که سیستم مدیریت پایگاه داده‌ها به کمک آن رکوردهای خاص را در یک فایل با سرعت زیاد پیدا می‌کند و به این ترتیب سرعت پاسخ به پرس و جوها افزایش می‌یابد. هر ساختار شاخص، حاوی رکوردهایی است که در هر رکورد یک مقدار کلیدی (کلید جستجو) و آدرس منطقی رکوردهای فایل داده‌ای نگهداری می‌شود.

اغلب سیستم‌های مدیریت پایگاه داده‌ها، از ساختار درخت برای ایجاد شاخص‌ها استفاده می‌کنند. عمق درخت، بیشترین تعداد سطوح از ریشه به برگ است. عمق ممکن است در مسیرهای مختلف از ریشه تا برگ متفاوت باشد. و همچنین عمق ممکن است در مسیرهای مختلف از ریشه تا برگ یکسان باشد، که در این شرایط با درخت متوازن و $B-Tree$ و B^+-Tree مواجه هستیم. هرچه درجه‌ی گره‌های درخت بیشتر شود، درخت پهن‌تر و کم‌عمق‌تری ایجاد می‌شود. از آنجاییکه زمان دسترسی در یک درخت، بیشتر وابسته به عمق درخت است تا پهنای آن، پس ساخت درخت پهن و کم‌عمق در ایجاد شاخص باعث افزایش سرعت جستجو می‌شود، ساختارهای $B-Tree$ و B^+-Tree درخت‌هایی با عمق کم و پهنای زیاد هستند. ساختار B^+ Tree index برای پاسخ به Range Query و Equality Query مفید است.

صورت سوال به این شکل است:

جدول T را در نظر بگیرید که روی ستون a، Clustered Index شده است. کدام مورد درست است؟

۱) اعمال سیاست شاخص‌گذاری، تاثیری بر حجم اطلاعات ذخیره‌شده بر روی دیسک ندارد.

گزینه اول نادرست است، زیرا تعریف و نگهداری شاخص موجب تحمیل سربار حافظه‌ای به سیستم می‌شود. شاخص بر روی دیسک نگهداری می‌شود و هنگام استفاده به حافظه اصلی آورده می‌شود، بنابراین اعمال سیاست شاخص‌گذاری، بر افزایش حجم اطلاعات ذخیره شده بر روی حافظه اصلی و هارد دیسک تاثیر دارد.

۲) با اعمال سیاست شاخص‌گذاری، پاسخ به Range Queryهای مرتبط به a، با سرعت

بیشتری انجام می‌شود.

گزینه دوم درست است، زیرا Clustered Index توسط ساختار B^+ Tree پیاده‌سازی می‌شود. در ساختار B^+ Tree جستجو و بازیابی نقطه‌ای(Equality Query) از مرتبه $O(Log_t n)$ و جستجو و بازیابی بازه‌ای(Range Query) از مرتبه $O(Log_t n+k)$است. بنابراین اعمال سیاست شاخص‌گذاری توسط ساختار B^+ Tree باعث می‌شود Range Queryها و Equality Queryهای مرتبط، با سرعت بیشتری انجام شود.

۳) با اعمال سیاست شاخص‌گذاری، پاسخ به Equality Queryهای مرتبط به a، با سرعت کمتری انجام می‌شود.

گزینه سوم نادرست است، زیرا Clustered Index توسط ساختار B^+ Tree پیاده‌سازی می‌شود. در ساختار B^+ Tree جستجو و بازیابی نقطه‌ای(Equality Query) از مرتبه $O(Log_t n)$ و جستجو و بازیابی بازه‌ای(Range Query) از مرتبه $O(Log_t n+k)$است. بنابراین اعمال سیاست شاخص‌گذاری توسط ساختار B^+ Tree باعث می‌شود Range Queryها و Equality Queryهای مرتبط، با سرعت بیشتری انجام شود.

۴) همه موارد درست هستند.

گزینه چهارم نادرست است، زیرا فقط گزینه دوم درست است.

۲- گزینه (۳) صحیح است.

دو هدف اصلی سیستم ذخیره و بازیابی اطلاعات در پایگاه داده‌ها، اول سرعت عملیات در ذخیره و بازیابی اطلاعات و دوم صرفه‌جویی در مصرف حافظه است. برای مثال کاهش افزونگی محتوایی (طبیعی) توسط نرمال‌سازی جداول منجر به کاهش میزان حافظه مصرفی می‌شود. عمل واکشی تک تک رکوردها وقت‌گیر است، برای رفع این عیب شاخص یا Index ابداع شد. برای اینکه بازیابی داده‌ها با سرعت و کارایی بیشتر صورت گیرد، از شاخص استفاده می‌شود. شاخص ساختمان داده‌ای است که سیستم مدیریت پایگاه داده‌ها به کمک آن رکوردهای خاص را در یک فایل با سرعت زیاد پیدا می‌کند و به این ترتیب سرعت پاسخ به پرس و جوها افزایش می‌یابد. هر ساختار شاخص، حاوی رکوردهایی است که در هر رکورد یک مقدار کلیدی (کلید جستجو) و آدرس منطقی رکوردهای فایل داده‌ای نگهداری می‌شود.

اغلب سیستم‌های مدیریت پایگاه داده‌ها، از ساختار درخت برای ایجاد شاخص‌ها استفاده می‌کنند. عمق درخت، بیشترین تعداد سطوح از ریشه به برگ است. عمق ممکن است در مسیرهای مختلف از ریشه تا برگ متفاوت باشد. و همچنین عمق ممکن است در مسیرهای مختلف از ریشه تا برگ یکسان باشد، که در این شرایط با درخت متوازن و $B-Tree$ و B^+-Tree مواجه هستیم. هرچه

درجه‌ی گره‌های درخت بیشتر شود، درخت پهن‌تر و کم‌عمق‌تری ایجاد می‌شود. از آنجاییکه زمان دسترسی در یک درخت، بیشتر وابسته به عمق درخت است تا پهنای آن، پس ساخت درخت پهن و کم‌عمق در ایجاد شاخص باعث افزایش سرعت جستجو می‌شود، ساختارهای $B-Tree$ و B^+-Tree درخت‌هایی با عمق کم و پهنای زیاد هستند. ساختار B^+ Tree index برای پاسخ به Range Query و Equality Query مفید است.

صورت سوال به این شکل است:

در مورد شاخص‌گذاری (Indexing)، کدام مورد نادرست است؟

۱) B^+ Tree index برای پاسخ به Range Query مفید است.

گزینه اول درست است، زیرا Clustered Index توسط ساختار B^+ Tree پیاده‌سازی می‌شود. در ساختار B^+ Tree جستجو و بازیابی نقطه‌ای(Equality Query) از مرتبه $O(Log_t\, n)$ و جستجو و بازیابی بازه‌ای(Range Query) از مرتبه $O(Log_t\, n+k)$ است. بنابراین اعمال سیاست شاخص‌گذاری توسط ساختار B^+ Tree باعث می‌شود Range Queryها و Equality Queryهای مرتبط، با سرعت بیشتری انجام شود.

۲) هر جدول می‌تواند حداکثر یک شاخص کلاستر شده داشته باشد.

گزینه دوم درست است، زیرا هر جدول می‌تواند حداکثر یک clustered Index داشته باشد.

۳) شاخص‌های از نوع Hash برای پاسخ به Range Query مفید هستند.

گزینه سوم نادرست است، زیرا در ساختار فایل مستقیم(تکنیک درهم‌سازی) جستجو و بازیابی نقطه‌ای(Equality Query) از مرتبه $O(1)$ است. دستیابی از مرتبه $O(1)$ به جدول پایه، به این معناست که بدون توجه، به اندازه جدول پایه، جهت دسترسی به یک رکورد دلخواه، همواره تعداد اندکی پیگرد نیاز است. در حالت ایده‌آل با یک حرکت می‌توان به رکورد مورد نظر دسترسی پیدا کرد. همچنین در ساختار فایل مستقیم(تکنیک درهم‌سازی) جستجو و بازیابی بازه‌ای(Range Query) می‌بایست برای تک تک اعضای بازه به طور مستقل تابع hash اجرا شود که این امر مستلزم صرف وقت و هزینه زیادی است. فایل مستقیم بی‌نظم است و امکان پردازش سریالی و ترتیبی را ندارد، زیرا فایل مستقیم بر حسب کلید مرتب نشده است. به عبارت دیگر به علت درهم بودن فایل، واکشی رکورد بعدی مشابه واکشی یک رکورد جدید است، پس امکان پردازش سریالی و ترتیبی در آن وجود ندارد، ساختار فایل مستقیم مناسب محیط‌هایی است که دستیابی سریع به رکوردها مورد نیاز است و پردازش‌ها ترتیبی و سریالی نباشد. بنابراین برای جستجو و بازیابی بازه‌ای(Range Query)، شاخص‌گذاری مرتب شده با ساختار B^+ Tree با مرتبه $O(Log_t\, n+k)$ مناسب‌تر است. و شاخص‌های از نوع Hash برای پاسخ به Range Query ها مفید نیست.

۴) هرچه تعداد شاخص‌های یک جدول بیشتر باشد، سرعت Insert در آن جدول کمتر می‌شود.

گزینه چهارم درست است، زیرا عملیات درج، حذف و بروزرسانی در جدولی که شاخص دارد یعنی خود جدول پایه، نسبت به جدولی که شاخص ندارد زمان بیشتری مصرف می‌کند و به تبع این عملیات کندتر خواهد بود. زیرا شاخص‌ها نیز همگام با جداول پایه خود نیاز به بروزرسانی دارند. بنابراین تنها روی ستون‌هایی شاخص ایجاد می‌گردد، که به تناوب روی آنها جستجو انجام می‌شود. بنابراین هرچه تعداد شاخص‌های یک جدول بیشتر باشد، سرعت Insert، Update و Delete در آن جدول کمتر می‌شود. اما خود شاخص عامل جستجو و بازیابی سریع اطلاعات از یک جدول پایه است.